Irie Pascal Programmer's Reference Manual

Author: Stuart King

Page: 1

TABLE OF CONTENTS

1.1 Preface

This is the Irie Pascal Programmer's Reference Manual, and describes the Irie Pascal language. This manual is not a tutorial and does not attempt to teach programming, if you are new to programming or new to Pascal you may need additional information. Fortunately there are many excellent books which teach programming in Pascal. If you decide to purchase a Pascal book, you should look for one that covers Standard Pascal. See Pascal book recommendations from the Irie Tools website.

See the Irie Pascal User's Manual for more information.

1.2 Compliance statement

Irie Pascal complies with the requirements of level 0 of ISO/IEC 7185, with the following exceptions: (Appendix B - Deviations from ISO/IEC 7185). ISO/IEC 7185 is the standard for the Pascal porgramming language published by the Internation Organization for Standardization.

NOTE: Irie Pascal compliance with ISO/IEC 7185 has not been formally certified by an external body.

2.1 What are language elements?

Language elemements are the symbols and separators that make up a Irie Pascal program. These language elements are:

- Character literals
- Reserved words
- Separators
- Compiler directives
- Special symbols
- String literals
- Integers
- Real numbers
- Identifiers

2.2 Character literals

Character literals are symbols that represent single characters, and can take one of three different forms:

1. enclosed by single quotes (')
2. enclosed by double quotes (")
3. prefixed by the pound sign (#)

Character Literals Enclosed By Single Quotes

Character literals can be formed by enclosing, the character being represented, in single quotes.
NOTE: Single quote characters, in character literals enclosed in single quotes, are represented by a pair of single quote characters. So

```
'A' represents the character A
```

and

```
'''' represents the character '
```

Character literals are case-sensitive so '**X**' is not equal to '**x**'.

NOTE: In Standard Pascal (ISO/IEC 7185) character literals must be enclosed in single quotes (').

Character Literals Enclosed By Double Quotes

Character literals can be formed by enclosing, the character being represented, in double quotes.
NOTE: Double quote characters, in character literals enclosed by double quotes, are represented as a pair of double quote characters. So

```
"A" represents the character A
```

and

```
"""" represents the character "
```

This second form of character literals, is an extension to Standard Pascal, and is provided mainly to make it easy to represent characters containing single quotes (i.e. you can enclose character literals representing single quotes with double quotes without using double single quotes like "'" instead of ''''.

Character Literals prefixed By The Pound Sign

Character literals can be formed by prefixing, the ordinal value of the character being represented, by the pound sign (#). So

```
#65represents the character with ordinal value 65 (i.e. A)
```

and

```
#9represents the character with ordinal value 9 (i.e. the tab character)
```

Examples

```
'A'    '+'    ' '    ''''    '"'
"A"    "+"    " "    "'"     """"
#65    #43    #32    #39     #34
```

Syntax

```
character-literal ='''string-element-one '''|
'"'string-element-two '"'|
'#'character-code

character-code =digit-sequence

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

digit-sequence =digit {digit }

printable-character =any character (including a space) that has a visual
representation.
```

```
string-element-one ='''' |printable-character

string-element-two ='""' |printable-character
```

NOTE: The production for printable-character doesn't use the usual notation because:

1. it's tedious to write out every possible printable character and
2. the definition for a printable character depends on the character set being used.

2.3 Reserved words

Reserved words (also called keywords) are symbols that have pre-defined and fixed meanings to the Irie Pascal compiler. The case of reserved words is usually not significant, so although the reserved words listed below are all in lowercase, by default the compiler will recognize reserved words regardless of the case of the letters. **NOTE**: There is a compiler option **(-p)** that will make reserved words and identifiers case-sensitive.

The reserved words recognized by the compiler are listed below (along with links to information describing how the reserved words are used):

NOTE: Reserved words added because of extensions to Standard Pascal are listed in **red**.

- **and** and (boolean operator), and (bitwise operator)
- **and_then** and_then operator
- **array** array types
- **begin** compound statement
- **case** record types, case statement
- **class** object types
- **const** constant definitions
- **div** div operator
- **do** while statement, for statement, with statement
- **downto** for statement
- **else** record types, case statement, if statement
- **end** record types, compound statement, case statement
- **file** file types
- **for** for statement
- **function** functions and procedures
- **goto** goto statement
- **if** if statement
- **in** in operator
- **label** labels
- **list** list types
- **mod** mod operator
- **nil** nil constant, pointer types
- **not** not (boolean operator), not (bitwise operator)
- **object** object types
- **of** array types, file types, list types, set types, record types, case statement
- **or** or (boolean operator), or (bitwise operator)
- **or_else** or_else operator
- **otherwise** record types, case statement
- **packed** array types, file types, list types, set types, record types
- **procedure** functions and procedures
- **program** Irie Pascal grammar
- **record** record types
- **repeat** repeat statement

- **set** set types
- **shl** shl operator
- **shr** shr operator
- **then** if statement
- **to** for statement
- **type** type definitions
- **until** repeat statement
- **var** variable declarations, function declarations, procedure declarations
- **while** while statement
- **with** with statement
- **xor** xor (boolean operator), xor (bitwise operator)

NOTE: Although reserved words look like identifiers, you can not use reserved words as identifiers.

2.4 Separators

The separators are spaces, tabs, end-of-lines, and comments, and they can appear anywhere in an Irie Pascal program without altering it's meaning, except that:

- At least one separator must appear between consecutive identifiers, reserved words, integers or real numbers.
- Separators can not occur inside any other language element (symbol).

Comments are sequences of characters enclosed in

```
{      or      (*
```

and

```
}      or      *)
```

As an extension to Standard Pascal, Irie Pascal also recognizes comments beginning with // (in which case the // and any text that follows on the same line is ignored). So for example the line below is a comment.

```
//rest of the text on the line
```

By default comments can not be nested (i.e. by default comments can not contain other comments). So comments like

```
(* outer (* inner comment *) comment *)
```

will be terminated at the first close comment marker, like below

```
(* outer (* inner comment *)
```

the last part

```
comment *)
```

will not be treated as part of the comment and will cause a syntax error.

There is a compiler option that causes the compiler to support nested comments. When this compiler option is enabled the compiler recognizes the end of comments only when the number of close comment markers matches the number of open comment markers. So the example comment above will terminate only after the second *).

Both open comment markers (* and { are considered to be equivalent, and both close comment markers *) and } are considered to be equivalent. So when nested comments are not enabled, the compiler will not recognize comments like the one below:

```
(* outer { inner comment } comment *)
```

Support for nested comments is disabled by default since in Standard Pascal comments do not nest.

Compiler directives are special kinds of comments, and are instructions to the compiler. See compiler directives for more information,

2.5 Compiler directives

Compiler directives are special kinds of comments, that give instructions to the compiler, and look like this:

```
OPEN_COMMENT $ NAME INFORMATION CLOSE_COMMENT
```

where **OPEN_COMMENT** is either

```
{      or      (*
```

and **CLOSE_COMMENT** is either

```
}      or      *)
```

and **NAME** is the name of the compiler directive and is on of the following:

- **B** - A flag directive that controls short-circuit evaluation. See Order of evaluation of operands of dyadic operators for more information about short-circuit evaluation. See also the -sc compiler option for the command-line compiler for more information (documented in the User's Manual).
- **I** - Both a flag directive that controls run-time I/O checking, and a value directive that specifies the name of a file to include. See traperrors procedure for more information.
- **P** - A flag directive that controls mandatory parenthesis mode. See the -p compiler option for the command-line compiler for more information (documented in the User's Manual).
- **R** - A flag directive that controls range checking. See the -r compiler option for the command-line compiler for more information (documented in the User's Manual).
- **S** - A flag directive that controls strict checking of string parameters passed by reference. See the -s compiler option for the command-line compiler for more information (documented in the User's Manual).
- **U** - A flag directive that controls checking for accessing undefined values. See the -u compiler option for the command-line compiler for more information (documented in the User's Manual).
- **V** - A flag directive that controls checking for accessing inactive record variants. See record types for more information on record variants. See also the -v compiler option for the command-line compiler for more information (documented in the User's Manual).
- **W** - A flag directive that controls whether the compiler issues warnings or not.

All of the compiler directive can be used as flag directives, and in this case the compiler directive instructs the compiler to turn an option on or off. When a compiler directive is used as a flag directive the **INFORMATION** part of the compiler directive is one of the following symbols:

- + : Turns the compiler option on.
- - : Turns the compiler option off

- . : Returns the compiler option to its previous setting before the last change. The compiler displays an error if there is no previous setting for the compiler option.

So for example the following compiler directive (*$R-*) turns range checking off.

The **I** compiler directive can be used to instruct the compiler to include a file inside the current file. In this case the **INFORMATION** part of the compiler directive is the name of the file to include.

```
For example:   (*$I windows.inc*)
```

The **W** compiler directive can be used to instruct the compiler to turn off or on individual warning messages. So for example:

```
(*$W44-*)
```

can be used to turn off the warning about "<X> is never used", before including a file which declares many identitifiers (some of which you don't intend to use) and

```
(*$W44+*)
```

can be used to turn this warning back on after the file is included.

2.6 Special symbols

The Irie Pascal compiler recognizes the following special symbols:

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

2.7 String literals

A string literal represents zero, two, or more characters and can take one of two different forms:

1. enclosed by single quotes (')
2. enclosed by double quotes (")

String Literals Enclosed By Single Quotes

String literals can be formed by enclosing, the characters being represented, in single quotes.
NOTE: Single quote characters, in string literals enclosed in single quotes, are represented by a pair of single quote characters. So

```
'Hello' represents the characters Hello
```

and

```
'Don''t' represents the characters Don't
```

String literals are case-sensitive so '**Hello**' is not equal to '**hello**'.

NOTE: In Standard Pascal (ISO/IEC 7185) string literals must be enclosed in single quotes (').

String Literals Enclosed By Double Quotes

String literals can be formed by enclosing, the character being represented, in double quotes.
NOTE: Double quote characters, in string literals enclosed by double quotes, are represented as a pair of double quote characters. So

```
"Hello" represents the characters Hello
```

and

```
"""Hello""" represents the characters "Hello"
```

This second form of string literals, is an extension to Standard Pascal, and is provided mainly to make it easy to represent characters containing single quotes (i.e. you can enclose string literals containing single quotes with double quotes without using double single quotes like "'" instead of ''')'.

Examples

```
''     '   '  'Don''t'        'Say "Hello"'     '!@#$%^&*()'
""     "   "  "Don't"         "Say ""Hello"""   "!@#$%^&*()"
```

Syntax

```
string-literal =empty-string |non-empty-string

empty-string ='''' |'""'

non-empty-string =
'''string-element-one string-element-one {string-element-one }'''|
'"'string-element-two string-element-two {string-element-two }'"'

printable-character =any character (including a space) that has a visual
representation.

string-element-one =''''|printable-character

string-element-two ='""'|printable-character
```

NOTE: The production for printable-character doesn't use the usual notation because:

1. it's tedious to write out every possible printable character and
2. the definition for a printable character depends on the character set being used.

2.8.1.1 What are integers?

Integers are numbers with no fractional part and can take one of the following three forms:

- Integers (Decimal notation)
- Integers (Hexadecimal notation)
- Integers (Binary notation)

Irie Pascal supports integers with values between

```
-2147483647  and  +4294967295
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
integer-number =decimal-integer |hexadecimal-integer |binary-integer
```

2.8.1.2 Integers (Decimal notation)

Integers in decimal notation (i.e. decimal integers) are formed using base 10, with the decimal digits **0** to **9** representing the values **zero** to **nine**.

Example

Here are some examples of valid decimal integers

```
100     0      7453    000005
```

Syntax

```
decimal-integer =digit-sequence

digit-sequence =digit {digit }

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

2.8.1.3 Integers (Hexadecimal notation)

Integers in hexadecimal notation (i.e. hexadecimal integers) are formed using base 16, with the decimal digits **0** to **9** representing the values **zero** to **nine** and the letters **A** to **F** representing the values **ten** to **fifteen**. Integers in hexadecimal notation are prefixed with the **$** symbol.

Example

Here are some examples of valid hexadecimal integers

```
$64     $0      $FF       $000aB4
```

Syntax

The syntax for hexadecimal integers is as follows:

```
hexadecimal-integer ='$'hex-digit-sequence

hex-digit-sequence =hex-digit {hex-digit }

hex-digit =digit |'a'|'b'|'c'|'d'|'e'|'f'
```

```
digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

2.8.1.4 Integers (Binary notation)

Integers in binary notation (i.e. binary integers) are formed using base 2, with the decimal digits **0** and **1** representing the values **zero** and **one**. Integers in binary notation are prefixed with the **%** symbol.

Example

Here are some examples of valid binary integers:

```
%01100110     %0      %11111111
```

Syntax

```
binary-integer ='%'binary-digit-sequence

binary-digit-sequence =binary-digit {binary-digit }

binary-digit ='0'|'1'
```

2.8.2.1 What are real numbers?

A Real number represents a numeric value with a fractional parts (the fractional part may be zero).

Irie Pascal supports real numbers with values between about **1e308** and about **-1e308**.

Examples

```
123.456
1.23456e2   which is also equal to 123.456
1.23456e02  which is also equal to 123.456
123456.0e-3 which is also equal to 123.456
7e-1        which is equal to 0.7
```

Syntax

```
real-number =
digit-sequence '.'fractional-part [exponent scale-factor ]|
digit-sequence exponent scale-factor

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

digit-sequence =digit {digit }

exponent ='e'

fractional-part =digit-sequence

scale-factor =[sign ]digit-sequence

sign ='-'|'+'
```

2.9.1 What are identifiers?

Identifiers are sequences of letters, digits, and underscores that start with a letter or an underscore (_). **NOTE:** Standard Pascal does not allow underscores in identifiers. By default identifiers are not case-sensitive so for example

```
grade and     Grade and     GRADE
```

are normally considered to be the same identifier. **NOTE:** There is a compiler option that will make reserved words and identifiers case-sensitive. Some programmers prefer case-sensitive languages since these languages consider identifiers that differ only in case to be different identifiers. This is often used to allow strongly related identifiers to have the same *spelling* (i.e. differ only in case).

Example

For example consider the code fragment below.

```
type
STUDENT = record
name : string;
address : string;
grade : integer;
end;
var
student : STUDENT;
```

The use of the same *spelling* for the variable **student** and its type **STUDENT** emphasize the connection between the two.

You should use this compiler option with caution (or not at all) since this feature of Irie Pascal is not supported by many (if any) other Pascal compilers.

Syntax

```
identifier =letter {letter |digit }

letter ='a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|
'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|
'u'|'v'|'w'|'x'|'y'|'z'|
'_'

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

2.9.2 Built-in identifiers

Built-in identifiers are automatically declared by the compiler. Unlike reserved words, you can declare your own identifiers with the same *spelling* as built-in identifiers, if you do then your declaration will override the declaration made by the compiler. So for example the following declaration

```
type
boolean = (no, yes);
```

will override the declaration created by the compiler

```
type
```

```
boolean = (false, true);
```

Some built-in identifiers are used by extensions to Standard Pascal and are therefore only declared by the compiler when the extensions are enabled.

2.9.3 User defined/declared identifiers

User-defined identifiers are created by the following declarations and definitions:

- Constant definitions
- Type definitions
- Variable declarations
- Function declarations
- Procedure declarations

2.9.4.1 What are directives?

Directives are special identifiers that can be used to give the compiler extra information about a declaration/definition. The compiler recognizes directives only at specific points in a declaration/definition, and this recognition is not affected by, and does not affect, any meaning the directive's identifiers may have outside of these specific points. So for example the forward directive is recognized even if the identifier **forward** has been declared to have some other meaning, and the use of the forward directive does not affect any meaning that **forward** has been declared to have.

The supported directives are listed below:

- The external directive.
- The forward directive.

2.9.4.2 The external directive

The **external** directive specifies that the function or procedure being declared is located outside of the program, and the function or procedure body is omitted from the declaraion. Currently the only external functions and procedures supported are functions and procedures located in Windows DLLs.

Example

The following external function declaration is taken from the **winuser.inc** include file, and declares the Windows MessageBox function.

```
function MessageBox(hWnd : _HWND; lpText, lpCaption : LPSTR; uType : UINT) : integer;
external dll='user32.dll' name='MessageBoxA' stdcall;
```

Syntax

The syntax for declaring external functions and procedures is given below (**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
function-declaration =function-heading ';'external-directive
```

```
procedure-declaration =procedure-heading ';'external-directive

dllname =string-literal

external-directive ='external' 'dll' '='dllname ['name' '='name ]['stdcall'|'cdecl']

function-heading ='function'identifier [formal-parameter-list ]':'result-type

name =identifier

procedure-heading ='procedure'identifier [formal-parameter-list ]

result-type =type-identifier
```

dllname is a string literal that is the name of the Windows DLL to call.

name is the name of the function or procedure in the DLL. If **name** is not specified then the name of the function or procedure in the DLL is assumed to be the same as the name of the function or procedure in the declaration.

stdcall specifies that the stdcall calling convention should be used (This is the default).

cdecl specifies that the cdecl calling convention should be used.

2.9.4.3 The forward directive

The **forward** directive specifies that the current function or procedure declaration is incomplete (the function or procedure block is omitted).

Example

The following simple example illustrates how to use the **forward** directive.

```
program forward(output);

procedure display(message : string); forward;

procedure DisplayHello;
begin
display('Hello');
end;

procedure display;
begin
writeln(message)
end;

begin
DisplayHello;
end.
```

NOTE: There ar two declarations for the procedure **display**. The first declaration for **display** uses the **forward** and omits the procedure block but declares the formal parameter **message**. The second declaration for **display** includes the procedure block which references the formal parameter **message** even though **message** is not declared in the second declaration. It would be an error or declare the formal parameter in the second declaration of **display**.

Syntax

The syntax for declaring forward functions and procedures is give below (**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
function-declaration =function-heading ';'forward-directive

procedure-declaration =procedure-heading ';'forward-directive

forward-directive ='forward'

function-heading ='function'identifier [formal-parameter-list ]':'result-type

procedure-heading ='procedure'identifier [formal-parameter-list ]

result-type =type-identifier
```

A second declaration for the function or procedure must appear later on in the program with the function or procedure block included but the formal parameter list omitted. The syntax for the second declaration of the function or procedure is given below (**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
function-declaration =function-identification ';'function-block

procedure-declaration =procedure-identification ';'procedure-block

block =declarative-part statement-part

declaration-group =
label-declaration-group |
constant-definition-group |
type-definition-group |
variable-declaration-group |
function-declaration |
procedure-declaration

declarative-part ={declaration-group }

function-block =block

function-identifier =identifier

procedure-block =block

procedure-identifier =identifier
```

3.1 What are labels?

Labels are markers, which you can place in front of statements so that the statements can be referenced by goto statements. Labels can be digit sequences or identifiers. In Standard Pascal labels must be digit sequences. Leading zeroes are not significant in labels, so "009" and "9" are the same label.

Example

```
//*************************************************************
//This program counts counts from 1 to 10 using goto
//*************************************************************
program count(output);
label write_count;
const
```

```
minimum = 1;
maximum = 10;
var
count : integer;
begin
count := minimum;
write_count: writeln(count);
count := count + 1;
if count <= maximum then goto write_count
end.
```

Labels must be declared (in a label declaration group) before they can be used.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
label-declaration-group ='label'label {','label }';'

label =digit-sequence |identifier

digit-sequence =digit {digit }

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

4.1 What are constants?

Constants can be character literals, string literals, integers, real numbers, or constant identifiers.

Constant identifiers are identifiers that have been associated with a constant by a constant definition or by an enumerated type definition.

Example

Here is an example of a constant definition group:

```
const
Space = ' ';
message = 'Hello';
first = 1;
last = 10;
DaysInYear = 365.25;
minint = -maxint;
linefeed = #10;
```

4.2.1 What are built-in constant identifiers

Built-in constant identifiers are automatically defined by the compiler. Unlike reserved words, you can declare your own identifiers with the same *spelling* as built-in constant identifiers, if you do then your declaration will override the definition made by the compiler. The built-in constant identifiers are listed below:

- appendmode
- dir_bit
- false

- feature_calldll
- feature_closedir
- feature_clrscr
- feature_delay
- feature_gotoxy
- feature_idispatch
- feature_intr
- feature_keypressed
- feature_mysql
- feature_odbc
- feature_opendir
- feature_popen
- feature_readdir
- feature_readkey
- feature_rewinddir
- feature_textbackground
- feature_textcolor
- feature_wherex
- feature_wherey
- grp_r
- grp_w
- grp_x
- maxbyte
- maxchar
- maxint
- maxword
- nil
- oth_r
- oth_w
- oth_x
- platform_dos
- platform_error
- platform_fbsd
- platform_linux
- platform_os2
- platform_solaris
- platform_solaris_sparc
- platform_win32
- readmode
- rsdefault
- rsdynamic
- rsforward
- rskeyset
- rsstatic
- sql_tc_all
- sql_tc_ddl_commit
- sql_tc_ddl_ignore
- sql_tc_dml
- sql_tc_none
- true
- usr_r
- usr_w
- usr_x
- writemode

4.2.2 appendmode

Description

appendmode is used with the built-in procedure open to specify that the file variable should be opened for writing, in such a way that all writes are made at the end of the associated file regardless of calls to the built-in procedure seek (i.e. the file should be opened in append mode).

Example

```
open(file_variable, 'filename', appendmode);
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.3 dir_bit

Description

dir_bit can be AND'd (see and operator (bitwise)) with the **mode** parameter returned by the built-in procedure getfilemode to determine if a file is a directory. **NOTE:** Directories are also called folders.

Example

The simple program below asks for a filename and then tests whether the file is a directory:

```
program isdirectory(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
if isdir then
writeln(s, ' is a directory')
else
writeln(s, ' is not a directory');
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.4 false

Description

false is the first enumerated constant defined by the built-in enumerated type boolean. The ordinal value of **false** is zero.

Portability

Operating Systems: All
Standard Pascal: Yes

4.2.5 feature_calldll - calling DLLs supported?

Description

feature_calldll is used with the built-in function supported to determine if the current platform supports calling external functions and procedures in DLLs.

Example

```
if supported(feature_calldll) then
begin
//Calling DLLs is supported so go ahead and make the calls.
end
else
begin
//Calling DLLs is not supported so do something else of abort.
end
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.6 feature_closedir - closedir supported?

Description

feature_closedir can be used to determine if the current platform supports the built-in procedure closedir.

Example

```
if (supported(feature_closedir)) and
(supported(feature_readdir)) and
(supported(feature_opendir)) then
```

```
begin
//open directory, retrieve filenames, and then close directory
end
else
begin
//Opening, reading, and closing directories is not supported so do
// something else of abort.
end
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.7 feature_clrscr - clrscr supported?

Description

feature_clrscr can be used to determine if the current platform supports the built-in procedure clrscr.

Example

```
if supported(feature_clrscr) then clrscr;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.8 feature_delay - delay supported?

Description

feature_delay can be used to determine if the current platform supports the built-in procedure delay.

Example

```
if supported(feature_delay) then delay(100);
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.9 feature_gotoxy - gotoxy supported?

Description

feature_gotoxy can be used to determine if the current platform supports the built-in procedure gotoxy.

Example

```
if not supported(feature_gotoxy) then
begin
writeln('gotoxy not supported on the current platform');
abort
end;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.10 feature_idispatch - COM/ActiveX objects supported?

Description

feature_idispatch can be used to determine if the current platform supports accessing COM objects using the IDISPATCH interface.

Example

```
program SendEmailWithCDONTS(output);
begin
obj : object;
begin
if supported(feature_idispatch) then
begin
obj := createobject('CDONTS.NewMail');
obj.send('from address', 'to address', 'subject', 'body', 1);
dispose(obj);
end
else
writeln('Can not send email')
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.11 feature_intr - intr supported?

Description

feature_intr can be used to determine if the current platform supports calling the built-in procedure intr.

Example

```
if not supported(feature_intr) then
begin
writeln('x86 interrupts not supported on the current platform');
abort
end;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.12 feature_keypressed - keypressed supported?

Description

feature_keypressed can be used to determine if the current platform supports calling the built-in function keypressed.

Example

```
if not supported(feature_keypressed) then
begin
writeln('keypressed not supported on the current platform');
abort
end;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.13 feature_mysql - MySQL databases supported?

Description

feature_mysql This constant can be used to determine if the current platform supports using MySQL databases.

Example

```
program Data(input, output);
var
conn : connection;

procedure DisplayError(msg : string);
begin
writeln('ERROR:',msg);
halt
end;

function GetConnectionString : string;
var
s, sDSN, sUSER, sPassword : string;

function GetStringValue(sPrompt : string) : string;
var
sValue : string;
begin
write(sPrompt);
readln(sValue);
GetStringValue := sValue
end;

begin
if supported(feature_odbc) then
begin
sDSN := GetStringValue('Enter Data Source Name (DSN):');
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'ODBC;DSN='+sDSN+';user='+sUser+';password='+sPassword;
end
else if supported(feature_mysql) then
begin
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'MYSQL;user="'+sUser+'";password="'+sPassword+'";socket="/tmp/mysql.sock"';
end
else
DisplayError('No database support detected');
GetConnectionString := s;
end;

begin
new(conn);
conn.open(GetConnectionString);
//
//Add code here to process database
//
conn.close;
dispose(conn);
end.
```

Portability

Operating Systems:All
Standard Pascal:No

4.2.14 feature_odbc - ODBC databases supported?

Description

feature_odbc can be used to determine if the current platform supports using ODBC to connect to databases.

Example

```
program Data(input, output);
var
conn : connection;

procedure DisplayError(msg : string);
begin
writeln('ERROR:',msg);
halt
end;

function GetConnectionString : string;
var
s, sDSN, sUSER, sPassword : string;

function GetStringValue(sPrompt : string) : string;
var
sValue : string;
begin
write(sPrompt);
readln(sValue);
GetStringValue := sValue
end;

begin
if supported(feature_odbc) then
begin
sDSN := GetStringValue('Enter Data Source Name (DSN):');
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'ODBC;DSN='+sDSN+';user='+sUser+';password='+sPassword;
end
else if supported(feature_mysql) then
begin
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'MYSQL;user="'+sUser+'";password="'+sPassword+'";socket="/tmp/mysql.sock"';
end
else
DisplayError('No database support detected');
GetConnectionString := s;
end;

begin
new(conn);
conn.open(GetConnectionString);
//
//Add code here to process database
//
conn.close;
dispose(conn);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.15 feature_opendir - opendir supported?

Description

feature_opendir can be used to determine if the current platform supports the built-in procedure opendir.

Example

```
if (supported(feature_closedir)) and
(supported(feature_readdir)) and
(supported(feature_opendir)) then
begin
//open directory, retrieve filenames, and then close directory
end
else
begin
//Opening, reading, and closing directories is not supported so do
// something else of abort.
end
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.16 feature_popen - popen supported?

Description

feature_popen can be used to determine if the current platform supports calling the built-in procedure popen.

Example

```
if not supported(feature_popen) then
begin
writeln('popen not supported on the current platform');
abort
end;
```

Portability

Operating Systems: All

Standard Pascal: No

4.2.17 feature_readdir - readdir supported?

Description

feature_readdir can be used to determine if the current platform supports the built-in procedure readdir.

Example

```
if (supported(feature_closedir)) and
(supported(feature_readdir)) and
(supported(feature_opendir)) then
begin
//open directory, retrieve filenames, and then close directory
end
else
begin
//Opening, reading, and closing directories are not supported so do
// something else of abort.
end
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.18 feature_readkey - readkey supported?

Description

feature_readkey can be used to determine if the current platform supports calling the built-in function readkey.

Example

```
if not supported(feature_readkey) then
begin
writeln('readkey not supported on the current platform');
abort
end;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.19 feature_rewinddir - rewinddir supported?

Description

feature_rewinddir can be used to determine if the current platform supports the built-in procedure rewinddir.

Example

```
if supported(feature_rewinddir) then rewinddir;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.20 feature_textbackground - textbackground supported?

Description

feature_textbackground can be used to determine if the current platform supports the built-in procedure textbackground.

Example

```
program color(output);
const
blue = 1;
white = 15;
begin
if supported(feature_textbackground) then textbackground(blue);
if supported(feature_textcolor) then textcolor(white);
writeln('white on blue');
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.21 feature_textcolor - textcolor supported?

Description

feature_textcolor can be used to determine if the current platform supports the built-in procedure textcolor.

Example

```
program color(output);
const
blue = 1;
white = 15;
begin
if supported(feature_textbackground) then textbackground(blue);
if supported(feature_textcolor) then textcolor(white);
writeln('white on blue');
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.22 feature_wherex - wherex supported?

Description

feature_wherex can be used to determine if the current platform supports the built-in function wherex.

Example

```
if supported(feature_wherex) then xpos := wherex;
if supported(feature_wherey) then ypos := wherey;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.23 feature_wherey - wherey supported?

Description

feature_wherey can be used to determine if the current platform supports the built-in function wherey.

Example

```
if supported(feature_wherex) then xpos := wherex;
if supported(feature_wherey) then ypos := wherey;
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.24 grp_r

Description

grp_r can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine if any user belonging to the same group as the owner of a file or directory has permission to read the file or directory.

Example

For example the simple program below asks for a filename and then displays the group permissions for the file.

```
program GroupPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
write('Group permissions :');
if (mode and grp_r) <> 0 then
write(' Read');
if (mode and grp_w) <> 0 then
write(' Write');
if (mode and grp_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.25 grp_w

Description

grp_w can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine if any user belonging to the same group as the owner of a file or directory has permission to write to the file or directory.

Example

For example the simple program below asks for a filename and then displays the group permissions for the file.

```
program GroupPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
write('Group permissions :');
if (mode and grp_r) <> 0 then
write(' Read');
if (mode and grp_w) <> 0 then
write(' Write');
if (mode and grp_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.26 grp_x

Description

grp_x can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine if any user belonging to the same group as the owner of a file or directory has permission to execute the file or search the directory.

Example

For example the simple program below asks for a filename and then displays the group permissions for the file.

```
program GroupPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
```

```
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
write('Group permissions :');
if (mode and grp_r) <> 0 then
write(' Read');
if (mode and grp_w) <> 0 then
write(' Write');
if (mode and grp_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.27 maxbyte

Description

maxbyte is equal to the largest byte value supported by Irie Pascal (i.e. **255**).

Portability

Operating Systems: All
Standard Pascal: No

4.2.28 maxchar

Description

maxchar is equal to the largest character value supported by Irie Pascal (i.e. **#255**).

Portability

Operating Systems: All
Standard Pascal: No

4.2.29 maxint

Description

maxint is equal to the largest supported integer value (i.e. **2,147,483,647**).

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

4.2.30 maxword

Description

maxword is equal to the largest supported word value (i.e. **4,294,967,295**).

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.31 nil

Description

nil is a pointer value which is compatible with all pointer types. This constant can be assigned to any pointer variable to indicate that the variable doesn't point anywhere, and any pointer variable can also be compared with **nil**.

As an extension to Standard Pascal, **nil** is also a object value which is compatible with all object types. This constant can be assiged to any object reference to indicate the the reference doesn't point at any object, and any object reference can be compared with **nil**.

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

4.2.32 oth_r

Description

oth_r can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine whether users not belonging to the same group as the owner of a file or directory has permission to read from the file or directory.

Example

For example the simple program below asks for a filename and then displays the file permissions for users who not belong to the same group as the owner of the file.

```
program OtherUsersPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
write('Other permissions :');
if (mode and oth_r) <> 0 then
write(' Read');
if (mode and oth_w) <> 0 then
write(' Write');
if (mode and oth_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.33 oth_w

Description

oth_w can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine whether users not belonging to the same group as the owner of a file or directory has permission to write to the file or directory.

Example

For example the simple program below asks for a filename and then displays the file permissions for users who not belong to the same group as the owner of the file.

```
program OtherUsersPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
write('Other permissions :');
if (mode and oth_r) <> 0 then
write(' Read');
if (mode and oth_w) <> 0 then
write(' Write');
if (mode and oth_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems:All
Standard Pascal:No

4.2.34 oth_x

Description

oth_x can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine whether users not belonging to the same group as the owner of a file or directory has permission to execute the file or search the directory.

Example

For example the simple program below asks for a filename and then displays the file permissions for users who not belong to the same group as the owner of the file.

```
program OtherUsersPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
```

```
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
write('Other permissions :');
if (mode and oth_r) <> 0 then
write(' Read');
if (mode and oth_w) <> 0 then
write(' Write');
if (mode and oth_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.35 platform_dos

Description

platform_dos can be compared with the value returned by the built-in function platform to determine whether the current platform is DOS (i.e. the interpreter running your program is from the DOS edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.36 platform_error

Description

platform_error is returned by the built-in function platform when the current platform is unknown (**NOTE:** This can not happen using the current or previous versions of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.37 platform_fbsd

Description

platform_fbsd can be compared with the value returned by the built-in function platform to determine whether the current platform is FreeBSD (i.e. the interpreter running your program is from the FreeBSD edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
```

```
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.38 platform_linux

Description

platform_linux can be compared with the value returned by the built-in function platform to determine whether the current platform is Linux (i.e. the interpreter running your program is from the Linux edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.39 platform_os2

Description

platform_os2 can be compared with the value returned by the built-in function platform to determine whether the current platform is OS/2 (i.e. the interpreter running your program is from the OS/2 edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.40 platform_solaris

Description

platform_solaris can be compared with the value returned by the built-in function platform to determine whether the current platform is Solaris/x86 (i.e. the interpreter running your program is from the Solaris/x86 edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.41 platform_solaris_sparc

Description

platform_solaris_sparc can be compared with the value returned by the built-in function platform to determine whether the current platform is Solaris/Sparc (i.e. the interpreter running your program is from the Solaris/Sparc edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.42 platform_win32

Description

platform_win32 can be compared with the value returned by the built-in function platform to determine whether the current platform is 32 bit Windows (i.e. your program is a Windows .EXE executable, or your program is a .IVM executable running under the interpreter from the Windows edition of Irie Pascal).

Example

The following simple program displays the current platform.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
```

```
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems:All
Standard Pascal:No

4.2.43 readmode

Description

readmode is used with the built-in procedure open to specify that the file variable should be opened in read-only mode, and with the built-in procedure popen to specify that the pipe should be opened for reading.

Examples

```
open(file_variable, 'filename', readmode); //open for reading

open(file_variable, 'filename', readmode+writemode); //open for reading and writing

open(file_variable, 'filename', readmode or writemode); //open for reading and
writing
```

Portability

Operating Systems:All
Standard Pascal:No

4.2.44 rsdefault

Description

rsdefault is used when opening a recordset (see recordset.open) to specify that the default type of recordset should be opened. If you are connected to a database with ODBC or if you are connected to a MySQL database then a forward type recordset is opened.

Example

```
rs.open(conn, sSQL, rsdefault);
```

In the example above

rs is the recordset object that will contain the opened recordset.

conn is the connection object containing an open connection to a database.

sSQL is a string variable containing the SQL statement that defines the recordset.

rsdefault specifies that the default type of recordset should be opened.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.45 rsdynamic

Description

rsdynmic is used when opening a recordset (see recordset.open) to specify that a dynamic recordset should be opened. Dynamic recordsets are scrollable (i.e. you can move both forwards and backwards through the records in the recordset). Database changes (i.e. record insertions, updates, or deletions) are visible to dynamic recordsets even after the recordset is opened. **NOTE:** Irie Pascal does not currently provide a way to move backwards through a recordset, so you can't take maximum advantage of dynamic recordsets. Also if the you are connected to a MySQL database then this constant is ignored and a forward-only recordset is opened instead.

Example

```
rs.open(conn, sSQL, rsdynamic);
```

In the example above

rs is the recordset object that will contain the opened recordset.

conn is the connection object containing an open connection to a database.

sSQL is a string variable containing the SQL statement that defines the recordset.

rsdynamic specifies that a dynamic recordset should be opened.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.46 rsforward

Description

rsforward is used when opening a recordset (see recordset.open) to specify that a forward-only recordset should be opened. Forward-only recordsets, as the name suggests, only allow you to move forwards. To return to previous record your program would have to close the recordset, reopen it, and skip to the record you are interested in. Database changes (i.e. record insertions, updates, or deletions) are **not** visible to forward-only recordsets after the recordset is opened.

Example

```
rs.open(conn, sSQL, rsforward);
```

In the example above

rs is the recordset object that will contain the opened recordset.

conn is the connection object containing an open connection to a database.

sSQL is a string variable containing the SQL statement that defines the recordset.

rsforward specifies that a forward-only recordset should be opened.

Portability

Operating Systems: All
Standard Pascal: No

4.2.47 rskeyset

Description

rskeyset is used when opening a recordset (see recordset.open) to specify that a keyset recordset should be opened. Keyset recordsets are scrollable (i.e. you can move both forwards and backwards through the records in the recordset). Some database changes (e.g. record updates) are visible to keyset recordsets after the recordset is opened, but others (e.g. record insertions and deletions) are not. **NOTE:** Irie Pascal does not currently support keyset recordsets, if you specify this constant then a dynamic recordset is opened instead, unless the database is MySQL in which case a forward-only recordset is opened.

Example

```
rs.open(conn, sSQL, rskeyset);
```

In the example above

rs is the recordset object that will contain the opened recordset.

conn is the connection object containing an open connection to a database.

sSQL is a string variable containing the SQL statement that defines the recordset.

rskeyset specifies that a keyset recordset should be opened.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.48 rsstatic

Description

rsstatic is used when opening a recordset (see recordset.open) to specify that a static recordset should be opened. Static recordsets are scrollable (i.e. you can move both forwards and backwards through the records in the recordset). Database changes (e.g. record inserts, updates, and deletions) are **not** visible to static recordsets after the recordset is opened. **NOTE:** Irie Pascal does not currently provide a way to move backwards through a recordset, so you can not take full advantage of static recordsets. Also if the database is MySQL then this constant is ignored and a forward-only recordset is opened instead.

Example

```
rs.open(conn, sSQL, rsstatic);
```

In the example above

rs is the recordset object that will contain the opened recordset.

conn is the connection object containing an open connection to a database.

sSQL is a string variable containing the SQL statement that defines the recordset.

rsstatic specifies that a static recordset should be opened.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.49 sql_tc_all

Description

If the transsupport property of a connection type object is equal to this constant (i.e. **sql_tc_all**) then the connection supports transactions, which can contain DML (data management) and/or DDL (data definition) statements.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.50 sql_tc_ddl_commit

Description

If the transsupport property of a connection type object is equal to this constant (i.e. **sql_tc_ddl_commit**) then the connection supports transactions, which can contain DML (data management) statements. DDL (data definition) statements in the transactions cause them to be committed.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.51 sql_tc_ddl_ignore

Description

If the transsupport property of a connection type object is equal to this constant (i.e. **sql_tc_ddl_ignore**) then the connection supports transactions, which can contain DML (data management) statements. DDL (data definition) statements in the transactions are ignored.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.52 sql_tc_dml

Description

If the transsupport property of a connection type object is equal to this constant (i.e. **sql_tc_dml**) then the connection supports transactions, which can contain DML (data management) statements. DDL (data definition) statements in the transactions cause an error.

Portability

```
Operating Systems:All
```

```
Standard Pascal:No
```

4.2.53 sql_tc_none

Description

If the transsupport property of a connection type object is equal to this constant (i.e. **sql_tc_none**) then the connection does not support transactions.

Portability

```
Operating Systems:All
Standard Pascal:No
```

4.2.54 true

Description

true is the second enumerated constant defined by the built-in enumerated type boolean. The ordinal value of **true** is one.

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

4.2.55 usr_r

Description

usr_r can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine whether the owner of a file or directory/folder has permission to read from the file or directory/folder.

Example

For example the simple program below asks for a filename and then displays the file permissions for the owner of the file.

```
program OwnerPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
```

```
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
if isdir then
writeln(s, ' is a directory')
else
writeln(s, ' is not a directory');
write('Owner permissions :');
if (mode and usr_r) <> 0 then
write(' Read');
if (mode and usr_w) <> 0 then
write(' Write');
if (mode and usr_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.56 usr_w

Description

usr_w can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine whether the owner of a file or directory/folder has permission to write to the file or directory/folder.

Example

For example the simple program below asks for a filename and then displays the file permissions for the owner of the file.

```
program OwnerPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
if isdir then
writeln(s, ' is a directory')
else
writeln(s, ' is not a directory');
write('Owner permissions :');
if (mode and usr_r) <> 0 then
write(' Read');
if (mode and usr_w) <> 0 then
```

```
write(' Write');
if (mode and usr_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.57 usr_x

Description

usr_x can be AND'd (see and operator (bitwise)) with the filemode retrieved by the built-in procedure getfilemode to determine whether the owner of a file or directory/folder has permission to execute the file or search the directory/folder.

Example

For example the simple program below asks for a filename and then displays the file permissions for the owner of the file.

```
program OwnerPermissions(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
if isdir then
writeln(s, ' is a directory')
else
writeln(s, ' is not a directory');
write('Owner permissions :');
if (mode and usr_r) <> 0 then
write(' Read');
if (mode and usr_w) <> 0 then
write(' Write');
if (mode and usr_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

Portability

Operating Systems: All
Standard Pascal: No

4.2.58 writemode

Description

writemode is used with the built-in procedure open to specify that the file variable should be opened in write-only mode, and with the built-in procedure popen to specify that the pipe should be opened for writing.

Examples

```
open(file_variable, 'filename', writemode); //open for writing

open(file_variable, 'filename', readmode+writemode); //open for reading and writing

open(file_variable, 'filename', readmode or writemode); //open for reading and
writing
```

Portability

Operating Systems: All
Standard Pascal: No

writemode

4.2.59.1 The feature constants

Irie Pascal supports several built-in constants that an application can pass to the built-in function supported to determine if particular features are available on the current platform.

One approuch is to have your application check on start-up to make sure all necessary features are available. If a necessary feature is not available your application could then report which feature is missing and abort.

Another approuch is to have your application run with reduced functionality if certain features are not available.

For example an application could pass the feature_odbc constant to the supported function to determine if ODBC is supported before attempting to use ODBC to connect to a database.

The feature constants are listed below:

- feature_calldll
- feature_closedir
- feature_clrscr
- feature_delay

- feature_gotoxy
- feature_idispatch
- feature_intr
- feature_keypressed
- feature_mysql
- feature_odbc
- feature_opendir
- feature_popen
- feature_readdir
- feature_readkey
- feature_rewinddir
- feature_textbackground
- feature_textcolor
- feature_wherex
- feature_wherey

4.2.60.1 The file mode constants

Irie Pascal supports three integer constants, which can be used with the built-in procedures open and popen, to specify the mode a file or pipe should be opened in.

The file mode constants are listed below:

- appendmode
- readmode
- writemode

4.2.61.1 The permission constants

Irie Pascal supports ten built-in integer constants which can be used with the built-in procedures mkdir and getfilemode to specify or identify read, write, and execute/search permissions, as well as to find out whether a file is a directory/folder.

The permission constants are listed below:

- dir_bit
- grp_r
- grp_w
- grp_x
- oth_r
- oth_w
- oth_x
- usr_r
- usr_w
- usr_x

Example

For example the simple program below asks for a filename and then displays the all the file permissions for the file.

```
program AllPermissions(input, output);
```

```
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
if isdir then
writeln(s, ' is a directory')
else
writeln(s, ' is not a directory');
write('Owner permissions :');
if (mode and usr_r) <> 0 then
write(' Read');
if (mode and usr_w) <> 0 then
write(' Write');
if (mode and usr_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
write('Group permissions :');
if (mode and grp_r) <> 0 then
write(' Read');
if (mode and grp_w) <> 0 then
write(' Write');
if (mode and grp_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
write('Other permissions :');
if (mode and oth_r) <> 0 then
write(' Read');
if (mode and oth_w) <> 0 then
write(' Write');
if (mode and oth_x) <> 0 then
if isdir then
write(' Search')
else
write(' Execute');
writeln;
end.
```

4.2.62.1 The platform constants

Irie Pascal supports several integer constants, which can be compared with the value returned by the built-in function platform to determine the platform your program is running on.

The platform constants are listed below:

- platform_dos
- platform_error
- platform_fbsd
- platform_linux
- platform_os2
- platform_solaris
- platform_solaris_sparc

- platform_win32

4.2.63.1 The recordset type constants

Irie Pascal supports several integer constants, which are used when opening a recordset (see recordset.open) to specify the type of recordset you want to open.

The recordset type constants are listed below:

- rsdefault
- rsdynamic
- rsforward
- rskeyset
- rsstatic

4.3.1 Constant definitions

Constant definitions are used to create constant identifiers (i.e. identifiers that are associated with a constant).

Example

```
const
Space = ' ';
message = 'Hello';
first = 1;
last = 10;
DaysInYear = 365.25;
minint = -maxint;
linefeed = #10;
```

Syntax

The syntax for constant definition groups is given below: (**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
constant-definition-group ='const'constant-definition ';'{constant-definition ';'}

constant =[sign ]integer-number |
[sign ]real-number |
[sign ]constant-identifier |
character-literal |
string-literal

constant-definition =identifier '='constant
```

5.1 What are types?

Types can be defined as groups of values that are related in some way. For example the type integer can be defined as the group of whole number values between -maxint and +maxint. Every constant, variable, parameter, and expression in your program is associated with a specific type.

Irie Pascal defines several built-in types that represent commonly used types. You can use these built-in types or you can define your own. You can even redefine the built-in types, but this is not recommended, because it makes your programs more difficult to understand. Someone reading your program is likely to assume that the built-in types have their normal definition.

Most types must be defined before they can be used, the exceptions to this rule are pointer types, and list types. Pointer and list types are allowed to be used before they are defined to allow self-referencing structures (i.e. structures that point at themselves e.g. linked lists) to be built.

5.2 Array types

Description

Array types define collections of values of the same type (called the array's component type) and associates an index type with each collections. Each member of an array's collection is called an array element and is identified by a value of the array's index type. The number of array elements in an array is fixed, and is equal to the number of values defined by the array's index type.

You can have arrays of arrays (i.e. an array's component type may itself be an array type). It is sometimes easier to think about arrays of arrays as multi-dimensional arrays (i.e. arrays defining a collection of values with an axis for each array component, where each element is identified by values from each component array's index type).

Irie Pascal supports fixed length strings as defined in Standard Pascal (i.e. fixed length strings, which are simply packed arrays of char, having a lower bound of one and an upper bound of two or greater). Fixed length strings have special properties in addition to the properties they share with other arrays. Fixed length strings, unlike other arrays, can be compared with the relational operators. Also fixed length strings, unlike other arrays, can be read from and written to text files as a unit using the built-in procedures read, readln, write, and writeln. **NOTE:** Irie Pascal also supports variable length string types.

NOTE: As a shorthand notation a multi-dimensional array type can be defined by listing the index types of each of the component arrays. In other words

```
array[1..10, char, boolean] of integer
```

is a shorthand notation for

```
array[1..10] of array[char] of array[boolean] of integer
```

Example

Here are some examples of array types:

```
array[1..10] of real
array[char] of boolean
array[-456..-450] of array[boolean] of integer
```

Syntax

The syntax for array types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
array-type ='array' [index-type-list ']' 'of'component-type

component-type =type-denoter

domain-type =type-identifier

index-type =ordinal-type

index-type-list =index-type {','index-type }

new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

ordinal-type =new-ordinal-type |ordinal-type-identifier

ordinal-type-identifier =identifier

type-denoter =type-identifier |new-type

type-identifier =identifier
```

5.3 Enumerated types

Description

Enumerated types define a finite group of ordered values. Each of these ordered values is associated with an identifier (an enumerated constant). Enumerated constants are ordered by the sequence in which they are defined, and they have consecutive ordinal numbers starting at zero. The built-in function ord can be used to retrieve the ordinal number of an enumerated constant.

Example

Here are some examples of enumerated types

```
(red, blue, green)
(low, medium, high)
(married, divorced, widowed, single)
```

In the examples above the first enumerated constant in each type (i.e. **red**, **low**, and **married**) have ordinal values of zero. The second enumerated constant in each type (i.e. **blue**, **medium**, and **divorced**) have ordinal values of one. The third enumerated constant in each type have ordinal values of two. And the fourth enumerated constant in the last type has ordinal value 3.

Syntax

The syntax for enumerated types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
enumerated-type ='('enumerated-constant-list ')'

enumerated-constant =identifier

enumerated-constant-list =enumerated-constant {','enumerated-constant }
```

5.4 File types

Description

File types define collections of values that usually have the same type (called the file type's component type). File type component types can not be file types or types which contain file types. There is a special file type named binary which has no component type, and can define collections of values with different types.

The collection of values defined by a file type is usually just called a file for short. Files are stored on external storage devices, unlike values of other types which are stored in memory. The main advantage of files over values of other types is that, files can persist after the program that created them is terminated. The main disadvantage of files compared to values of other types is that, accessing files is usually much slower than accessing values of other types (this is because accessing external storage devices is usually much slower than accessing memory).

In theory files can store an unlimited number of values, however in practice the number of values stored in files is limited by the amount of available space on the external storage devices used to store the files. The operating system controlling the external storage devices will also have a limit on the size of files, but this limit is usually very large and does not affect most programs.

You can add to, and change the values of, a file using the following built-in procedures:

- seek
- put
- write
- writeln

You can retrieve values from a file using the following built-in procedures:

- get
- read
- readln
- seek

Before you can add values to, change the existing values of, or retrieve values from, a file you have to associate it with a file variable. **NOTE:** This process is sometimes referred to as *opening* the file, or *opening* the file variable. See append, open, reset, and rewrite.

When you have finished adding values to, changing the existing values of, or retrieving values from, a file you should disassociate the file from the file variable. **NOTE:** This process is sometimes referred to as *closing* the file. See close. It is not necessary to close files if the program is about to terminate.

Although files are stored on external storage devices, the operating system will usually temporarily store portions of open files in memory to speed up access. However this means that changes made to files can be lost if the program crashes. If you want to make sure that the changes made to a file are stored on an external storage device, then use the built-in procedure flush.

Example

Here are some examples of file types

```
file of integer;
file of char;
```

Syntax

The syntax for defining new file types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
file-type ='file' 'of'component-type

component-type =type-denoter

domain-type =type-identifier

new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

type-denoter =type-identifier |new-type

type-identifier =identifier
```

5.5 Integral types

Description

The integral types are one of the following types or subranges of one the following types:

- address
- byte
- integer
- shortint
- shortword
- word

NOTE: In Standard Pascal the integral types are integer and subranges of integer.

5.6 List types

Description

As an extension to Standard Pascal, Irie Pascal supports list types. List types define ordered collections of values of a single type (called the list's component type). Each value in a list type's collection is identified by it's position in the collection. Unlike file types the values in a list type's collection are NOT persistent (i.e. they are not stored when your program is not running).

The size of the collection of values defined by a list type (i.e. the number of values in the collection) is potentially unlimited. List types do not impose any limit on the size of their collections. In practice the maximum size of a list type's collection is determined by the amount of memory available to your program.

Example

Here are some examples of list types

```
list of integer;
list of char;
```

Below is a simple example program that uses a list to store and sort random integer values.

```
program listexample(output);
var
l : list of integer;
i : integer;

procedure SortList;
var
i, temp : integer;
swapped : boolean;
begin
writeln('Sorting...');
//Bubble sort the list
repeat
swapped := false;
for i := 1 to length(l)-1 do
begin
if l[i] > l[i+1] then
begin
temp := l[i];
l[i] := l[i+1];
l[i+1] := temp;
swapped := true;
end
end;
until not swapped;
end;


begin
randomize;  //initialize random number generator
new(l);
//Insert 100 random integer values into the list
for i := 1 to 100 do
```

```
insert(random(1000), l);
//Sort the values in the list
SortList;
//Output the values in the list
for i := 1 to length(l) do
writeln(l[i]);
dispose(l);
end.
```

Syntax

The syntax for defining new list types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
list-type ='list' 'of'component-type

component-type =type-denoter

domain-type =type-identifier

new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

type-denoter =type-identifier |new-type

type-identifier =identifier
```

5.7 Ordinal types

Description

Ordinal types define discrete groups of ordered values. Each value of a given ordinal type has a unique predecessor (except the first value defined by that type). Each value of a given ordinal type has a unique successor (except the last value defined by that type).

The ordinal types are enumerated types, integral types, and their subranges.

5.8 Pointer types

Description

Pointer types define dynamic collections of values, each of which either identifies an instance of the pointer type's domain type or is the special value nil which is guaranteed **not** to identify any instance. Instances of a pointer type's domain type are called identified variables. The values of a pointer type's collection can be stored in variables, associated with the pointer type, called pointer variables. The act of using the value stored in a pointer variable to access the identified variable is called dereferencing the pointer variable. It is an error to deference a pointer variable which contains the special value nil.

A pointer variable can be passed to the built-in procedure new, which will create an identified variable of the same type as the pointer variable's domain type, and store the value identifying the identified variable in the pointer variable. The value identifying the identified variable is also added to the collection of values defined by the pointer variables type. It is an error if an identified variable can not be created because there is not enough available memory.

A pointer variable can also be passed to the built-in procedure dispose, which will destroy the identified variable identified by the value stored in the pointer variable. The value stored in the pointer variable is also removed from the collection of values defined by the pointer variable's type. It is an error to use dispose on a pointer variable containing the special value nil, or containing a value that is not in the collection of values defined by the pointer variable's type.

Example

Here are some examples of pointer types:

```
^integer
^real
```

Syntax

The syntax for defining new pointer types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
new-pointer-type ='^'domain-type |'@'domain-type

domain-type =type-identifier
```

5.9 Record types

Description

Record types define collections of values. A record type may be empty (i.e. define a collection with no values). Each member of a record type's collection is called a field and is identified by a field identifier, with the possible exception of the variant selector which sometimes does not have a field identifier. Each member in a record's collection has a particular type (which can be a record type).

Sometimes groups of fields in a record type are mutually exclusive, in which case you can use variant parts in the record type to specify mutually exclusive groups of fields (these mutually exclusive groups are called variants). Each variant part contains a variant selector, and the value of this variant selector determines which variant if any is active. It is an error to access a field of a variant which is not active.

Example

Here are some examples of valid record types:

```
record end
```

The record type above is empty.

```
record
name : string;
grade : integer;
end
```

The record type above has two fields. The first field is identified by the field identifier **name**, and this field is of type string. The second field is identified by the field identifier **grade**, and this field is of type integer.

Given the type below

```
type
clothes = (shoe, pants, shirt);
```

the following record type is valid

```
record
price : real;
case kind : clothes of
shoe : ( size : integer; );
pants : ( waist, length : integer );
shirt : ( neck : integer; sleeve : integer )
end
```

and contain seven fields

price, **kind**, **size**, **waist**, **length**, **neck**, and **sleeve**.

The fields **price** and **kind** are always present and the value of the field **kind** determine which of the variants if any is present. The variants are

- **(size : integer;)** - which is active when **kind** equals **shoe**
- **(waist, length : integer)** - which is active when **kind** equals **pants**
- **(neck : integer; sleeve : integer)** - which is active when **kind** equals **shirt**

Syntax

The syntax for defining new record types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
record-type ='record'field-list 'end'

case-constant =ordinal-constant

case-constant-list =case-specifier { ','case-specifier }

case-specifier =case-constant [ '..'case-constant ]

domain-type =type-identifier
```

```
field-list =[
fixed-part ';'variant-part [';']|
fixed-part [';']|
variant-part [';']|
]

fixed-part =record-section {';'record-section }

identifier-list =identifier {','identifier }

new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

record-section =identifier-list ':'type-denoter

tag-field =identifier

tag-type =ordinal-type-identifier

type-denoter =type-identifier |new-type

type-identifier =identifier

variant =case-constant-list ':''('field-list ')'

variant-body =variant-list [[';']variant-part-completer ]|variant-part-completer

variant-list =variant {';'variant }

variant-part ='case'variant-selector 'of'variant-body

variant-part-completer =('otherwise'|'else')(field-list )

variant-selector =[tag-field ':']tag-type
```

5.10 Set types

Description

Set types define values which specify whether individual elements (known not suprisingly as set elements) are present or not. The group of possible set elements of a particular set type form an ordinal type, known as the set's base type. Set values do not specify whether set elements are present in any partucular order, and set elements can not be present more than once. All set types define the empty set (i.e. the set value specifying that none of the set elements are present).

For example the following set type

```
set of boolean
```

defines four values

1. [] - neither false nor true are present.
2. [false] - false is present but not true
3. [true] - true is present but not false
4. [false, true] - both false and true are present.

Irie Pascal uses two different representations for values of set types depending on the range of the set's base type. For example suppose you have the following set type:

```
set of T
```

then if **T's** range is less than or equal to 256 then the set is represented as a bit set, otherwise the set is represented using an array representation of fixed size.

Example

Here are some examples of set types

```
set of char
set of boolean
```

Syntax

The syntax for defining new set types is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
set-type ='set' 'of'base-type

base-type =ordinal-type

new-ordinal-type =enumerated-type |subrange-type

ordinal-type =new-ordinal-type |ordinal-type-identifier

ordinal-type-identifier =identifier
```

5.11 String types

Description

Fixed-Length Strings

Irie Pascal supports fixed length strings as defined in Standard Pascal (i.e. fixed length strings, which are simply packed arrays of char, having a lower bound of one and an upper bound of two or greater). Fixed length strings have special properties in addition to the properties they share with other arrays. Fixed length strings, unlike other arrays, can be compared with the relational operators. Also fixed length strings, unlike other arrays, can be read from and written to text files as a unit using the built-in procedures read, readln, write, and writeln.

Variable Length Strings

As an extension to Standard Pascal, Irie Pascal supports variable length strings defined using the built-in types **string** and **cstring**. Variable length strings define sequences of characters. The maximum number of characters allowed in a variable length string is fixed and is specified when the string type is defined.

Variable length strings defined with **string** are stored as length prefixed strings (i.e. the actual number of characters in the string is stored in a byte or integer in front of the string). A byte is used to store the actual length of the string, if the maximum number of characters allowed in the string is small enough to be stored in a byte (i.e. <= 255). An integer is used to store the actual length of the string, if the maximum number of characters allowed in the string is too big to fit in a byte (i.e. > 255).

Variable length strings defined with **cstring** are stored as NULL-terminated strings.

Example

For example to create a variable length string type called **name** with a maximum length of 80 use

```
name = string[80];
```

or

```
name = string(80);
```

To create a variable length string type called **phone** with a maximum length of 255 use

```
phone = string;
```

or

```
phone = string[255];
```

or

```
phone = string(255);
```

In most case it is best to use the built-in type **string** to create variable length string types. However variable length string types defined with **cstring** are useful when the values of a variable length string are going to be passed to or from external functions/procedures written in the **C** programming language. For example this simple hello world program uses two variables of **cstring** types to pass values to the Windows API function **MessageBox**.

```
program winhello;
(*$I winuser.inc *)
const
NULL_HANDLE = 0;
CR = 13;

var
csText, csCaption : cstring;
RetVal : integer;

begin
csText := 'Hello world!!'+chr(CR)+'Press Cancel when done';
csCaption := 'Irie Pascal Windows Hello World Program';
repeat
RetVal := MessageBox(NULL_HANDLE, addr(csText), addr(csCaption), MB_OKCANCEL +
MB_ICONEXCLAMATION);
```

```
until RetVal = IDCANCEL;
end.
```

Syntax

The syntax for defining new variable string types is given below (see array types for the sytax for defining new fixed length strings):

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
string-type =pascal-string-type |c-string-type

c-string-type ='cstring'[max-string-length ]

max-string-length ='['integral-constant ']'|
'('integral-constant ')'

pascal-string-type ='string'[max-string-length ]
```

where **max-string-length** is between 1 and 1048576, and specifies the maximum length of the string. If **max-string-length** is not specified, the default maximum length (i.e. 255) is used.

5.12 Subrange types

Description

Subrange types define a contiguous subset of the values of an ordinal type. This ordinal type is known as the subrange type's host type and the values of a subrange type have the same type as its host type.

Example

Here are some examples of subrange types:

```
-100..+100
true..true
'a'..'z'
```

Syntax

The syntax for defining new subrange types is give below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
subrange-type =constant '..'constant
```

5.13 Variant Types

Description

The **variant** type is a special type that can contain values of the following types:

- boolean
- byte
- integer
- real
- string
- word

The **variant** type is the type of the field property of recordset objects, and is only used when accessing values from recordset fields (which can have a variety of types). **Variant** values are automatically converted to appropriate types, whenever they are used in expressions. So you can't use this type to declare variables or parameters.

The idea behind the **variant** type is to make database handling more robust, by isolating programs to some extent from the specific type of the fields in databases.

5.14.1 What are object types?

Description

Object types define dynamic collections of values, each of which either identifies an instance of an object described by the object type or is the special value nil, which is guaranteed **not** to identify any instance of an object. The values in object type collections are stored in variables, associated with the object type, called object references. These object references can be used to refer to or access the object instance identified by the value stored in the object reference.

Object instances have properties (which are similiar to the fields in records), and methods. Method are functions or procedures that define what the object can do and also what can be done to the object.

Irie Pascal does not currently fully support object types, in particular Irie Pascal does not allow you to define your own object types. You are restricted to using the built-in object types connection and recordset, or using the generic object type.

The built-in procedure new can be used to create an instance of one of the built-in object types, and store the value identifying the instance in an object reference. This also adds the identifying value to the collection of values defined by the built-in object type associated with the object reference. The built-in procedure new can **not** be used to create instances of generic object type, since generic object type do not describe what an object instance "looks" like (new wouldn't know how much memory to allocate).

The built-in function createobject can be used to create an instance of a generic object type, and returns the value identifying the instance. The value returned by createobject is usually stored in a generic object reference. **NOTE:** The built-in function createobject can **not** currently be used to create an instance of any of the built-in object types.

The built-in procedure dispose should be used to destroy instances of the built-in object types created with new, when you have finished using the instances. This also removes the value identifying the destroyed instance from the collection of values defined by the built-in object type.

The built-in procedure dispose should also be used to destroy instances of generic objects created with createobject. This also removes the value identifying the destroyed instance from the collection of values defined by the generic object type.

Example

For example the simple program below illustrates how to use the built-in procedures new and dispose with object types.

```
program Data(input, output);
var
conn : connection;

procedure DisplayError(msg : string);
begin
writeln('ERROR:', msg);
halt
end;

function GetConnectionString : string;
var
s, sDSN, sUSER, sPassword : string;

function GetStringValue(sPrompt : string) : string;
var
sValue : string;
begin
write(sPrompt);
readln(sValue);
GetStringValue := sValue
end;

begin
if supported(feature_odbc) then
begin
sDSN := GetStringValue('Enter Data Source Name (DSN):');
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'ODBC;DSN='+sDSN+';user='+sUser+';password='+sPassword;
end
else if supported(feature_mysql) then
begin
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'MYSQL;user="'+sUser+'";password="'+sPassword+'";socket="/tmp/mysql.sock"';
end
else
DisplayError('No database support detected');
GetConnectionString := s;
end;

begin
new(conn);
conn.open(GetConnectionString);
//
//Add code here to process database
//
conn.close;
dispose(conn);
end.
```

Syntax

The syntax for accessing an object property is give below

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
property-designator =object-variable '.'property-specifier

object-variable =variable-access

property-identifier =identifier

property-specifier =property-identifier |'('property-string ')'

property-string =string-expression
```

where **string-expression** is an expression of type string which evaluates to the name of the property. The preferred form of a property reference is just to use the property name, the second form where a string expression is used is provided for the cases where the property name is a keyword. You can see this in the cdonts example below in which the property named **to** could not be specified as

```
objMail.to
```

since **to** is a keyword, instead this property is specified using the second form like so

```
objMail.('to')
```

The syntax for calling object method functions and procedures is given below:

```
function-method-designator =object-variable '.'function-method-identifier [actual-
parameter-list ]

procedure-method-statement =procedure-method-specifier [actual-parameter-list ]

actual-parameter =expression |variable-access |
procedure-identifier |function-identifier

actual-parameter-list ='('actual-parameter {','actual-parameter }')'

function-method-identifier =identifier

object-variable =variable-access

procedure-method-identifier =identifier

procedure-method-specifier =object-variable '.'procedure-method-identifier
```

5.14.2 Generic objects

Description

Generic object types are not any particular object type, but are instead any object type associated with an instance of an object created with the built-in function createobject. Currently createobject can only create instances of COM objects, and only those COM objects which support late-binding (i.e. the COM objects must export the IDISPATCH interface, or in other words the COM objects must be OLE automation servers).

Example

Below is a simple program that uses two generic object references **objExplorer** and **app** to access some of the COM objects exposes by Internet Explorer, and to start Internet Explorer and navigate to the IrieTools website.

```
program Explorer;
var
objExplorer : object;
app : object;
begin
objExplorer := CreateObject('InternetExplorer.Application');
app := objExplorer.application;
app.navigate('http://www.irietools.com', 1, , ,);
app.quit;
dispose(app);
dispose(objExplorer);
end.
```

Below is a more useful example program that illustrates how to send an email using the CDONTS.NewMail COM object that is installed with Microsoft IIS. You can use a similiar program to send email if your webserver uses IIS.

```
program cdonts;
const
NORMAL = 1;
var
objMail : object;
FromAddress, ToAddress : string;
begin
write('Enter the from address:');
readln(FromAddress);
write('Enter the to address:');
readln(ToAddress);
objMail := CreateObject('CDONTS.NewMail');
objMail.from := FromAddress;
objMail.('to') := ToAddress;
objMail.subject := 'Testing...';
objMail.Body := 'This is a test message';
objMail.Importance := NORMAL;
objMail.Send( , , , , );
dispose(objMail);
end.
```

The program could also be simplified as follows

```
program cdonts;
const
NORMAL = 1;
var
objMail : object;
FromAddress, ToAddress : string;
begin
write('Enter the from address:');
readln(FromAddress);
write('Enter the to address:');
readln(ToAddress);
objMail := CreateObject('CDONTS.NewMail');
objMail.Send(FromAddress, ToAddress, 'Testing...', 'This is a test message', NORMAL);
dispose(objMail);
end.
```

Syntax

The syntax for defining a new generic object type is given below:

```
object-type ='object'|'class'
```

The keywords **object** and **class** can be used interchangably for now, however in later versions of Irie Pascal, which will more fully support Object Pascal, these keywords may produce slightly different effects. There appears to be two specifications for Object Pascal. The first specification is simplier and uses the keyword **object** and appears to be most popular on the Apple Mac. While the second specification is more sophisticated and uses the keyword **class**. Later versions of Irie Pascal may attempt to support both specifications of Object Pascal.

5.15.1 The built-in type identifiers

Built-in type identifiers are automatically defined by the compiler. Unlike reserved words, you can define your own identifiers with the same *spelling* as built-in type identifiers, if you do then your definition will override the definition made by the compiler. The built-in type identifiers are listed below:

- address type
- binary type
- boolean type
- byte type
- char type
- connection type
- dir type
- double type
- error type
- filename type
- integer type
- real type
- recordset type
- registers type
- regtype type
- shortint type
- shortword type
- single type
- text type
- word type

5.15.2 address type

Description

address is the type identifier for a built-in type, whose values are operating system addresses. Irie Pascal usually generates and uses virtual machine addresses which are not meaningful to external functions/procedures. When passing a variable by reference to an external function/procedure, Irie Pascal will automatically convert the address of the variable from a virtual machine address to an operating system address. This automatic conversion is usually enough to allow you to call external functions/procedures. However in some cases additional work is necessary, for example when the external function/procedure expects to pass operating system addresses back to you, or accepts parameters whose type can vary. In those cases it may be necessary to declare the variable or the component of the variable that will contain the operating system address, to be of type **address**.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.3 binary type

Description

binary is the type identifier for a built-in file type which does not have a component type (i.e. an untyped file type). Normally files can only contain values of the same type, which is called the file's component type, however this restriction does not apply to untyped file types. Values of almost any type can be written to and read from untyped file types. Also, because untyped file types do not have component types, there are no buffer variables associated with untyped file types.

Example

For example below is a silly little program that illustrates how to use **binary** file types.

```
program silly;
var
f : binary;
begin
rewrite(f, 'silly.dat');
write(f, 100, -67.56, false);
end.
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.4 boolean type

Description

boolean is the type identifier for a built-in enumerated type with two values false and true defined as below:

```
boolean = (false, true);
```

The ordinal value of false is zero and the ordinal value of true is one. The conditional statement:

- if

and the looping statements:

- while
- repeat

use expressions of boolean type. The operators:

- and
- and_then
- not
- or
- or_else

operate on expressions of boolean type.

Example

For example below is a simple program that illustrates using the built-in type **boolean**:

```
program Teenager(input, output);
var
age : integer;
teenager : boolean;
begin
write('How old are you?');
readln(age);
teenager := (age >= 13) and (age <= 19);
if teenager then
writeln('Congratulations!!! You are a teenager')
else
writeln('Sorry. You are not a teenager')
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

5.15.5 byte type

Description

byte is the type identifier for a built-in 8 bit ordinal type, whose values are the integers between 0 and 255 (maxbyte).

Example

Below is a simple program that writes all possible values of **byte** to a file.

```
program bytes;
var
f : file of byte;
```

```
b : byte;
begin
rewrite(f, 'bytes.dat');
for b := 0 to maxbyte do
write(f, b)
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.6 char type

Description

char is the type identifier for a built-in ordinal type with values of the current character set.

Example

Below is a simple program that illustrates using **char** to do some file processing.

```
(*
This program is used to count the number of vowels and the total number of
characters in the input file.
NOTE: This program makes use of two features of Irie Pascal
1) If the user does not specify a command-line parameter to match the
program argument 'f' then 'f' is automatically assigned an empty string
as its filename.
2) If a file variable with an empty string as its filename is reset then
the standard input file is associated with the file variable.

The net effect of these two features is that if you run this program and
don't supply a program argument like "ivm vowels" then the program reads
from the standad input file. If you run this program with a program argument
like "ivm vowels filename" then the program will open "filename" and read
from it.
*)
program vowels(f, output);
var
f : file of char;
tot, vow : integer;
c : char;
begin
reset(f);
tot := 0;
vow := 0;
while not eof(f) do
begin
read(f, c);
case c of
'a', 'e', 'i', 'o', 'u',
'A', 'E', 'I', 'O', 'U'
: vow := vow + 1;
otherwise
end;
tot := tot + 1;
end;
writeln('Total characters read = ', tot);
writeln('Vowels read = ', vow)
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

5.15.7 dir type

Description

dir is the type identifier for a built-in type whose values are directory handles. **NOTE:** Directories are also called folders.

Example

Below is a simple program that lists all the directories in the current directory.

```
program listdir(output);
var
d : dir;
filename : string;

function IsDir(name : filename) : boolean;
var
mode : integer;
begin
getfilemode(name, mode);
IsDir := (mode and dir_bit) <> 0;
end;

begin
opendir(d, '.');  { Open current directory }
repeat
readdir(d, filename);  { Get next file in directory }
{ if the next file exists and it is a directory then list it }
if (filename <> '') and (IsDir(filename)) then
writeln(filename);
until filename = '';
closedir(d) { Close directory }
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.8 double type

Description

double is a synonym for real.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.9 error type

Description

error is the type identifier for a built-in record type which describes a trappable run-time error.

The "error" type is defined as follows:

```
error = record
number : integer;
name : string[16];
description : string[255];
end;
```

where **number** is a numeric error code and uniquely identifies the error, **name** is an alphnumeric error code (not all errors have a **name**), **description**" is text describing the error.

When a trappable error occurs it is placed in an **error** record, and the record is appended to the built-in list variable errors.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.10 filename type

Description

filename is the type identifier for a built-in string type which is the recommended type for variables used to store file and directory/folder names.

Example

Below is a simple program that lists the names of all the files in the current directory and uses variables of type **filename** to store the names of the files.

```
program listfiles(output);
var
d : dir;
filename : string;

function IsDir(name : filename) : boolean;
var
mode : integer;
begin
getfilemode(name, mode);
IsDir := (mode and dir_bit) <> 0;
end;

begin
opendir(d, '.');  { Open current directory }
repeat
readdir(d, filename);  { Get next file in directory }
{ if the next file exists and it is NOT a directory then list it }
if (filename <> '') and (not IsDir(filename)) then
writeln(filename);
until filename = '';
closedir(d) { Close directory }
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.11 integer type

Description

integer is the type identifier for a built-in ordinal type, whose values are 32-bit signed integers between -2147483647 and +2147483647 (maxint).

Example

Below is a simple example program that prints the odd numbers between 1 and 24.

```
program OddNumbers(output);
const
first = 1;
last = 24;
var
i : integer;

function IsOdd(i : integer) : boolean;
begin
IsOdd := (i mod 2) <> 0;
end;

begin
for i := first to last do
if IsOdd(i) then writeln(i);
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

5.15.12 real type

Description

real is the type identifier for a built-in type whose values are 64 bit real numbers.

Example

For example the rather pointless program below uses the **real** type to repeatedly divide a number into thirds.

```
program thirds(output);
const
initial = 37.6;
lowest= 0.05;
var
value : real;
begin
value := initial;
repeat
writeln(value, value:12:3);
value := value / 3
until value < lowest
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

5.15.13 registers type

Description

registers is the type identifier for a built-in variant record type, which is used to store the x86 processor registers before and after calls to the built-in procedures intr and msdos.

The record layout is given below:

```
regtype = (byteregs, wordregs)

registers =  record
case regtype of
byteregs : ( al, ah, afill1, afill2 : byte;
bl, bh, bfill1, bfill2 : byte;
cl, ch, cfill1, cfill2 : byte;
dl, dh, dfill1, dfill2 : byte;
);
wordregs : eax, ebx, ecx, edx, esi, edi, cflag : word;
end;
```

Example

The simple program below clears the screen by using the built-in procedure intr to call the BIOS INT $10 interrupt.

```
program cls;
var
regs : registers;
begin
fill(regs, 0);
regs.eax := $0002;
intr($10, regs)
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.14 regtype type

Description

regtype is the type identifier for a built-in enumerated type, which is part of the variant record type, which is used to store the x86 processor registers before and after calls to the built-in procedure intr.

The variant record layout is given below:

```
regtype = (byteregs, wordregs)

registers =  record
case regtype of
byteregs : ( al, ah, afill1, afill2 : byte;
bl, bh, bfill1, bfill2 : byte;
cl, ch, cfill1, cfill2 : byte;
dl, dh, dfill1, dfill2 : byte;
);
wordregs : eax, ebx, ecx, edx, esi, edi, cflag : word;
end;
```

Example

The simple program below clears the screen by using the built-in procedure intr to call the BIOS INT $10 interrupt.

```
program cls;
var
regs : registers;
begin
fill(regs, 0);
```

```
regs.eax := $0002;
intr($10, regs)
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.15 shortint type

Description

shortint is the type identifier for a built-in ordinal type, whose values are signed 16-bit integers between -32767 and +32767.

This type is provided to make it easier to call calling external functions/procedures that expect and/or produce signed 16-bit integers. **NOTE:** Some functions/procedures in the Windows API expect and/or produce signed and unsigned 16-bit integers.

Portability

Operating Systems: All
Standard Pascal: No

5.15.16 shortword type

Description

shortword is the type identifier for a built-in ordinal type, whose values are unsigned 16-bit integers between 0 and 65535.

This type is provided to make it easier to call calling external functions/procedures that expect and/or produce signed 16-bit integers. **NOTE:** Some functions/procedures in the Windows API expect and/or produce signed and unsigned 16-bit integers.

Portability

Operating Systems: All
Standard Pascal: No

5.15.17 single type

Description

single is the type identifier for a built-in type whose values are 32 bit real numbers.

This type is provided to make it easier to call calling external functions/procedures that expect and/or produce 32-bit real numbers. **NOTE:** Some functions/procedures in the Windows API expect and/or produce 32-bit real numbers.

Portability

Operating Systems: All
Standard Pascal: No

5.15.18 text type

Description

text is the type identifier for a built-in file type, whose values are sequences of lines. A line is a sequence of characters and every line in a **text** file is terminated by an end-of-line character except possibly the last line (see Termination of all lines in text files).

Example

The simple program below writes a short message to a text file.

```
program message(fOut);
const
OutputFileName='message.txt';
var
fOut : text;
begin
rewrite(fOut, OutputFileName);
writeln(fOut, 'This file is generated by a very simple program');
writeln(fOut, 'which was written to help explain the "text" file type.');
close(fOut);
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

5.15.19 word type

Description

word is the type identifier for a built-in ordinal type, whose values are 32-bit unsigned integers between 0 and 4294967295 (maxword).

Example

Below is a simple example program that prints the odd numbers between 1 and 24.

```
program OddNumbers(output);
const
first = 1;
last = 24;
var
w : word

function IsOdd(w : word) : boolean;
begin
IsOdd := (w mod 2) <> 0;
end;

begin
for w := first to last do
if IsOdd(w) then writeln(w);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.1 The connection type

Description

connection is the type identifier for a built-in object type, whose values identify instances of connection objects. As the name suggests, instances of connection objects are used to connect to database engines, currently connections can only be made via ODBC or to MySQL databases.

The connection object type has the following methods:

- connection.close
- connection.execute
- connection.open
- connection.selectdatabase

The connection object type has the following properties:

- connection.databasename
- connection.dbmsname

- connection.dbmsname
- connection.dmver
- connection.odbcver
- connection.procedures
- connection.readonly
- connection.transactions
- connection.transsupport

The built-in procedure new is used to create instances of connection objects, and the built in procedure dispose is used to detroy instances of connection objects.

Example

The simple program below illustrates how to connect to a database engine using the connection object type.

```
program Data(input, output);
var
conn : connection;

procedure DisplayError(msg : string);
begin
writeln('ERROR:',msg);
halt
end;

function GetConnectionString : string;
var
s, sDSN, sUSER, sPassword : string;

function GetStringValue(sPrompt : string) : string;
var
sValue : string;
begin
write(sPrompt);
readln(sValue);
GetStringValue := sValue
end;

begin
if supported(feature_odbc) then
begin
sDSN := GetStringValue('Enter Data Source Name (DSN):');
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'ODBC;DSN='+sDSN+';user='+sUser+';password='+sPassword;
end
else if supported(feature_mysql) then
begin
sUser := GetStringValue('Enter user id:');
sPassword := GetStringValue('Enter password:');
s := 'MYSQL;user="'+sUser+'";password="'+sPassword+'";socket="/tmp/mysql.sock"';
end
else
DisplayError('No database support detected');
GetConnectionString := s;
end;

begin
new(conn);
conn.open(GetConnectionString);
//
//Add code here to process database
```

```
//
conn.close;
dispose(conn);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.2 connection.close

Description

The **close** method procedure of the connection object type is used to close an open connection to a database engine.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.close;
```

will close the connection.

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.3 connection.execute

Description

The **execute** method procedure of the connection object type is used to send an SQL statement to a database engine through an open connection.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.execute('create database test');
```

will send the SQL statement

```
create database test
```

to the database engine at the other end of the connection.

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.4 connection.open

Description

The **open** method procedure of the connection object type is used to open a connection to a database engine.

Example

For example, the simple program below illustrates how to call the **open** method. **NOTE:** The actual connection strings, you will use in your own programs will probably differ from the one used in the program below.

```
program Connect;
var
conn : connection;
begin
new(conn); //Create instance of connection object before using it
if supported(feature_odbc) then
conn.open('ODBC;DSN=test;user=sa;password=')
else if supported(feature_mysql) then
conn.open('MYSQL;user="sa";password="";socket="/tmp/mysql.soc"')
//
//Add code here to process database
//
conn.close;
dispose(conn); //Destroy instance of connection object when finished
end.
```

Parameter

The only parameter to this procedure is the connection string, which is an expression of type string.

NOTE: Before an attempt is made to actually open a connection, the connection string is processed to determine what kind of connection to open and what are the parameters to use for the connection. The connection string is processes as follows:

The connection string must be prefixed by a connection type specifier, which is separated from the actual connection string by a semi-colon (;).

First the connection type specifier is separated from the connection string. Next the connection type specifier is compared with **ODBC**, in such a way that case is **not** significant, and if there is a match then an ODBC connection is to be opened. Since case is not significant in this comparision the connection type specifier could be lowercase (as in **odbc**), uppercase (as in **ODBC**), or mixed case. If the connection type specifier does not match **ODBC**, then it is compared with **MYSQL**, again in such a way that case is NOT significant. If the connection type specifier matches **MYSQL** then a MySQL connection is to be opened. It is an error if the connection string is not prefixed by a connection type specifier or if the connection type specifier does not match either **ODBC** or **MYSQL**.

Opening ODBC Connections

Next if an ODBC connection is to be opened then the connection string must be in one of the following two forms:

```
odbc-connection-string =
odbc-connection-string-1 |
odbc-connection-string-2 |

odbc-connection-string-1 =
dsn-parm ;uid-parm ;pwd-parm [;[driver-specific-text ] ]

dsn-parm =DSN =name

uid-parm =UID =uid

pwd-parm =PWD =[password ]

odbc-connection-string-2 =name ;[uid] ;[password ]
```

where [] indicate optional parameters.

```
For example  DSN=test;UID=sa;PWD=
```

When the connection string is in the first form then it is passed, without further processing, to the ODBC API function **SQLDriverConnect** to open the connection.

When the connection string is in the second form then the **name**, **id**, and **password** parameters are extracted from the connection string, if present, and passed to the ODBC API function **SQLConnect** to open the connection. **NOTE:** The first form of the connection string is the recommended form, support for the second form is provided for completeness only.

Opening MySQL Connections

If a MySQL connection is to be opened then the connection string must be in the following form:

```
mysql-connection-string =mysql-parameter-list

mysql-parameter-list =mysql-parameter ;mysql-parameter-list |empty

mysql-parameter =
mysql-host-parameter |
mysql-user-parameter |
mysql-password-parameter |
mysql-database-parameter |
mysql-port-parameter |
mysql-socket-parameter |
mysql-compress-parameter

mysql-host-parameter =host = "host-name "

mysql-user-parameter =user = "user-name "

mysql-password-parameter =password = "password "

mysql-database-parameter =database = "database-name "
```

```
mysql-port-parameter =port =port-number

mysql-socket-parameter =socket = "socket "

mysql-compress-parameter =compress =boolean-value

boolean-value =yes|no|true|false

For example    user="testuser";database="testdb";socket="/tmp/mysql.soc";
```

The connection parameters are extracted from the connection string and passed to the MySQL C API function **mysql_real_connect** to open the connection.

The effect of each of the parameters is described below:

- The **mysql-host-parameter** specifies the hostname or IP address of the MySQL database server. If **mysql-host-parameter** is not specified or if **host-name** is an empty string or **local-host** then the connection is opened to the local MySQL server over a UNIX socket.
- The **mysql-user-parameter** specifies the username used to connect to the MySQL database server. If the **mysql-user-parameter** is not specified or if **user-name** is an empty string then the UNIX login name of the person running the program is used.
- The **mysql-password-parameter** specifies the password of the user who will be connected to the database server. If the **mysql-password-parameter** is not specified or if **password** is an empty string then the connection is rejected if the user actually has a password.
- The **mysql-database-parameter** specifies the initial database selected when the connection is opened. If the **mysql-database-parameter** is not specified or if **database** is an empty string then no initial database is selected. In which case you must call the connection.selectdatabase method later on to select a database.
- The **mysql-port-parameter** specifies the port used to remotely connect to a MySQL database server over TCP. If the **mysql-port-parameter** is not specified or if **port-number** is 0 then the default port is used.
- The **mysql-socket-parameter** specifies the filename of the UNIX socket used to connect to a MySQL database server on the local machine. If the **mysql-socket-parameter** is not specified or if **socket** is an empty string then the default socket is used.
- The **mysql-compress-parameter** specifies that compression is to be used when communicating with the MySQL database server. If the **mysql-compress-parameter** is not specified then compression is not used.

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.5 connection.selectdatabase

Description

The **selectdatabase** method procedure is used to make a particular database the default database for an open connection. **NOTE:** If you want to retrieve the name of the current database for a connection, use the connection.databasename property.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.selectdatabase('test');
```

will make **test** the default database for the connection.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.6 connection.databasename

Description

The **databasename** property of the connection object type contains the name of the default database for the connection. **databasename** is a read-only property of type string. **NOTE:** If you want to change the current default database for a connection, call the connection.selectdatabase method.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.databasename
```

is equal to the default database for the connection referenced by **conn**.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.7 connection.dbmsname

Description

The **dbmsname** property of the connection object type, contains the name of the database engine connected to. **NOTE:** If the connection is to a MySQL database then this property always constains **MySQL**. **dbmsname** is a read-only property of type string.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.dbmsname
```

is equal to the named of the database engine referenced by **conn**.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.8 connection.dbmsver

Description

The **dbmsver** property of the connection object type, contains the version of the database engine connected to. **dbmsver** is a read-only property of type string.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.dbmsver
```

is equal to the version of the database engine referenced by **conn**.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.9 connection.dmver

Description

The **dmver** property of the connection object type, contains the version of the ODBC driver manager connected to. **NOTE:** If the connection is to a MySQL database then this property always constains an empty string (""). **dbmver** is a read-only property of type string.

Example

For example, if **conn** is a connection object reference with an open connection to an ODBC database engine then the following:

```
conn.dmver
```

is equal to the version of the ODBC driver manager.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.10 connection.odbcver

Description

The **odbcver** property of the connection object type, contains the version of ODBC that the ODBC driver manager conforms to. **NOTE:** If the connection is to a MySQL database then this property always constains an empty string (""). **odbcver** is a read-only property of type string.

Example

For example, if **conn** is a connection object reference with an open connection to an ODBC database engine then the following:

```
conn.odbcver
```

is equal to the version of ODBC that the ODBC driver manager conforms to.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.11 connection.procedures

Description

The **procedures** property of the connection object type, indicates whether or not stored procedures are supported by the database engine connected to. **procedures** is a read-only property of type boolean.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.procedures
```

indicates whether stored procedures are supported by the database engine.

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.12 connection.readonly

Description

The **readonly** property of the connection object type, indicates whether or not the connection to the database engine is readonly. **NOTE: readonly** is a read-only property of type boolean.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.readonly
```

indicates whether or not the connection referenced by **conn** is read-only.

Portability

Operating Systems: All
Standard Pascal: No

5.15.20.13 connection.transactions

Description

The **transactions** property of the

connection object type indicates connection object type, indicates whether or not transactions are supported by the database engine connected to. **NOTE: transactions** is a read-only property of type boolean.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.transactions
```

indicates whether transactions are supported by the database engine.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.20.14 connection.transsupport

Description

The **transsupport** property of the connection object type, indicates the level of transaction support provided by the database engine connected to. **NOTE: transsupport** is a read-only property of type integer.

Example

For example, if **conn** is a connection object reference with an open connection to a database engine then the following:

```
conn.transsupport
```

contains the level of transaction support provided by the database engine. The possible values of this property are:

- sql_tc_none - Transactions are not supported.
- sql_tc_dml - Transactions containing Data Manipulation Language (DML) statements are supported. Transactions containing Data Definition Language (DDL) statements cause an error. DML statements include SELECT, INSERT, UPDATE, DELETE. DDL statements include CREATE, and DROP.
- sql_tc_all - Transactions containing both DML and DDL statements are supported.
- sql_tc_ddl_commit - Transactions containing DML statements are supported. DDL statements encountered in transactions cause the transactions to be committed.
- sql_tc_ddl_ignore - Transactions containing DML statements are supported. DDL statements encountered in transactions are ignored.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.21.1 The recordset type

Description

recordset is the type identifier for a built-in object type whose values identify instances of recordset objects. Instances of recordset objects are used to retrieve groups of records from open database connections.

The **recordset** object type has four methods:

- recordset.close
- recordset.moreresults
- recordset.movenext
- recordset.open

The recordset object type has two properties:

- recordset.eof
- recordset.field

Example

The code fragment below show a typical use of a recordset. NOTE: The code fragment is not a complete program and makes the following assumptions:

1. **conn** is an connection object reference for an open connection object.
2. **conn** is connected to a database with a table named **customer**.
3. The **customer** table has the fields **last_name** and **first_name**.

```
new(rs);
rs.open(conn, 'select last_name, first_name from customer', rsforward);
while not rs.eof
begin
writeln(rs.field('last_name'), ', ', rs.field('first_name'));
rs.movenext;
end;
rs.close;
dispose(rs);
```

First the code fragment creates an instance of a recordset object. Next the recordset object is used to open a forward type recordset. Next the code loops through all the records contained in the recordset printing out the contents of the fields **last_name** and **first_name**. Next the recordset is closed. Finally the instance of the recordset object is destroyed.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.21.2 recordset.close

Description

The **close** method procedure of the recordset object type is used to close an open recordset.

Example

For example, if you have a recordset object reference **rs** that refers to an open recordset then the following call:

```
rs.close;
```

will close the recordset.

Portability

Operating Systems: All
Standard Pascal: No

5.15.21.3 recordset.moreresults

Description

The **moreresults** method procedure of the recordset object type is used to to move to the next recordset contained by an open recordset. **NOTE:** This method should not to be confused with the movenext method which moves to the next record in the current recordset. This method is most often used when calling stored procedures that return multiple recordsets.

Example

For example if you open a recordset like so

```
rs.open(conn, 'EXEC procName')
```

and the stored procedure **procName** returns multiple recordsets then the recordset object will be positioned at the first record of the first recordset. The call

```
rs.movenext;
```

would move to the second record in the first recordset, but the call

```
rs.moreresults
```

would move to the first record in the second recordset.

Portability

Operating Systems: All
Standard Pascal: No

5.15.21.4 recordset.movenext

Description

The **movenext** method procedure of the recordset object type is used to move to the next record in the current recordset of an open recordset object.

Example

For example, if you have a recordset object reference **rs** that refers to an open recordset then the following call:

```
rs.movenext
```

will move to the next record in the recordset.

Portability

```
Operating Systems:All
Standard Pascal:No
```

5.15.21.5 recordset.open

Description

The **open** method procedure of the recordset object type is used to open a recordset. In order to call this method you need a recordset object reference, a connection object reference to open connection object, as well as the SQL statement which will return the recordset or recordsets, that will be referenced by the open recordset object.

Example

The code fragment below show a typical use of a recordset. **NOTE:** The code fragment is not a complete program and makes the following assumptions:

1. **conn** is an connection object reference for an open connection object.
2. **conn** is connected to a database with a table named **customer**.
3. The **customer** table has the fields **last_name** and **first_name**.

```
new(rs);
rs.open(conn, 'select last_name, first_name from customer', rsforward);
while not rs.eof
begin
writeln(rs.field('last_name'), ', ', rs.field('first_name'));
rs.movenext;
end;
rs.close;
dispose(rs);
```

Parameters

1. This first parameter to this method is a reference to an open connection object.
2. The second parameter to this method is an expression of type string that evaluates to a SQL statement that will return one or more recorsets.
3. The third parameter to this method is an expression of type integer that evaluates to the same value as one of the built-in recordset type constants (you should just use one of the recordset type constants directly). If **recordset-type** is not specified then the default recordset type is assumed.

Portability

Operating Systems: All
Standard Pascal: No

5.15.21.6 recordset.eof

Description

The **eof** property of the recordset object type indicates whether or not the recordset is currently at end-of-file. **NOTE: eof** is a read-only property of type boolean.

Example

For example if you have a recordset object reference **rs** with an open recordset then

```
rs.eof
```

indicates, whether the current recordset referenced by **rs** is at end-of-file.

It is an error to call the movenext method of a recordset object when the recordset is at end-of-file.

Portability

Operating Systems: All
Standard Pascal: No

5.15.21.7 recordset.field

Description

The **field** property of the recordset object type is used to retrieve the value of a field of the current record in the current recordset. The isnull function can be used to test if the value of the field is NULL. **NOTE: field** is a read-only property of type variant.

Currently, this property is read-only so you can not use it to change the contents of the record fields (i.e. you can't use this property to change the database). If you want to change the database call the execute method of the connection object type and pass in a SQL UPDATE statement.

Example

For example, if you have a recordset object reference **rs** with an open recordset and the records in the recordset have a field named **last_name** then

```
rs.field('last_name')
```

constains the contents of the field **last_name** for the current record in the recordset.

NOTE: Since this property is used so often, you can leave out the name of the property (i.e. **field**). So for example

```
rs.('last_name')
```

is equivalent to

```
rs.field('last_name')
```

Portability

Operating Systems: All
Standard Pascal: No

5.16.1 Type definitions

Description

Type identifiers are identifiers that have been associated with a type using a type definition. Type definitions must be placed in type definition groups.

Example

Here is an example of a type definition group:

```
type
symbols = record
name : string;
value : integer;
end;
SymbolTable = list of symbols;
color = (red, green, blue);
cardinal = 0..maxint;
IntegerList = array[1..100] of integer;
characters = set of char;
GenericObject = object;
```

Syntax

The syntax for type definition groups is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
type-definition-group ='type'type-definition ';'{type-definition ';'}

domain-type =type-identifier

new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

type-definition =identifier '='type-denoter

type-denoter =type-identifier |new-type

type-identifier =identifier
```

6.1 What are variables?

Description

A variable is made up of one or more contiguous memory locations, and is used to store values of a particular type. It is a common shorthand to refer to the type of the values that can be stored in a variable, as the variable's type. The number of contiguous memory locations that make up a variable (also called the variable's size) will depend on the variable's type. It is a common shorthand to refer to the value stored in a variable, as the value of the variable, or the variable's value.

Variables are dynamic, and can be both created and destroyed. When a variable is created it's initial value (i.e. it's value before a value has been explicitly stored in it) is said to be *undefined*. It is an error to retrieve an *undefined* value from a variable.

Values are stored inside variables by referencing the variables in the left-hand side of assignment statements, and the values stored inside variables are retrieved by referencing the variables in expressions. **NOTE:** The method used to reference a variable depends on what kind of variable it is.

Variables can be grouped into the following two categories:

1. named variables
2. anonymous variables

6.2 Named variables

Description

Named variables are introduced in variable declarations, which define the type of each variable, and associate an identifier (i.e. a *name*) with each variable.

Named variables are created when the block containing their variable declaration is activated, and destroyed when the activation of the block is terminated. In the case of variables introduced in the variable declarations of the main program block, this means that these variables will be created when the program is activated, and destroyed when the program is terminated. In the case of variables introduced in the variable declarations of function or procedure blocks, this means that these variables will be created when the function or procedure is activated, and destroyed when the function or procedure is terminated.

Named variables are referenced simply by using their *names* (i.e. the identifiers associated with the variables).

Example

For example below is a simple program that introduces an integer variable, associates the variable identifier **n** with the new variable, stores 100 in the the variable, retrieves the value stored in the variable, and displays the value on the screen.

```
program number(output);
var
n : integer; (* declare variable 'n' of type 'integer' *)
begin
n := 100;        (* Store 100 in the variable *)
writeln(n)       (* Retrieve the value in the variable and display it on the screen *)
end.
```

6.3 Anonymous variables

Description

Anonymous variables are **not** associated with identifiers. Anonymous variables can be grouped into the following four sub-categories:

1. Buffer variables.
2. Identified variables (see pointer variables).
3. Instance variables (see object variables).
4. Selected variables (see list variables).

6.4 Array variables

Description

Array variables are used to store values of array types (i.e. collections of values of the same type). The type of each value in an array is called the array's *component type*. An array's *component type* may be any type, including an array type, so it is possible to define arrays of arrays (also called multi-dimensional arrays).

A reference to an array variable is a reference to the entire array as a unit, and can be used to both store array values into, and retreive array values from the array variable.

The number of values in each array is fixed, and each value in the array is identified by a value of the array's index type (which is specified when the array is defined). Each value in an array is stored in a seperate component of the array called an array element (or an indexed variable). Each element of an array can be referenced individually, in order to store values into, and retrieve values from specific elements. See the syntax section below for details.

Example

Below are some examples of array variable declarations.

```
scores : array[1..200] of integer;
name : array[IndexType] of real;
```

```
students : array[1..MAX_STUDENTS] of student;
grades : array[1..10] of array[1..5] of integer;
grades : array[1..10, 1..5] of integer;
```

NOTE: These examples assume that **IndexType** is a previously defined ordinal type, **MAX_STUDENTS** is a previously defined constant of integral type, and **student** is a previously defined type. Also the two declarations shown for the array **grades** are equivalent.

Here is a simple example program that generates some random numbers, stores them in an array, sorts the numbers, and then prints the numbers out.

```
program sort(output);
const
max = 20;
var
numbers : array[1..max] of integer;

procedure GenerateRandomNumbers;
var
i : 1..max;
begin
for i := 1 to max do
numbers[i] := random(100)+1;
end;

procedure SortNumbers;
var
i : 1..max;
swapped : boolean;
temp : integer;
begin
repeat
swapped := false;
for i := 1 to max-1 do
if numbers[i] > numbers[i+1] then
begin
temp := numbers[i];
numbers[i] := numbers[i+1];
numbers[i+1] := temp;
swapped := true;
end;
until not swapped;
end;

procedure PrintNumbers;
var
i : 1..max;
begin
for i := 1 to max do
writeln(i, numbers[i]);
end;

begin
randomize;
GenerateRandomNumbers;
SortNumbers;
PrintNumbers;
end.
```

Syntax

See array types for the syntax for defining new array types.

The syntax for referencing array elements (i.e. indexed variables) is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
indexed-variable =indexed-variable-array |indexed-variable-string

array-variable =variable-access

index-expression =expression

indexed-variable-array =array-variable '['index-expression {','index-expression }']'
```

Where **array-variable** is a reference to an array variable, and **index-expression** is an expression which evaluates to one of the values specified by the array's index type.

6.6 File variables

Description

File variables are used to process files (i.e. create, destroy, change, or retrieve values from files). The term file is a shorthand way of saying *collection of values defined by a file type*.

The following built-in functions and procedures operate on file variables:

- eof
- eoln
- filepos
- filesize
- append
- assign
- close
- flush
- get
- open
- page
- put
- rawread
- rawwrite
- read
- readln
- rename
- reset
- rewrite
- seek
- write
- writeln

Most file variables are associated with buffer variables. File variables of type text are associated with buffer variables of type char. File variables of other types (except binary) are associated with buffer variables whose type is the same as the component type of the file variables type. File variables of type binary do not have a particular component type, and so they do not have buffer variables.

Example

The simple example program below uses a file variable (**f**) to create a file called **numbers.dat** and write the numbers 1 to 10 to the file.

```
program numbers;
var
f : file of integer;
i : integer;
begin
assign(f, 'numbers.dat');
rewrite(f);
for i := 1 to 10 do
write(f, i);
end.
```

Syntax

The syntax for referencing file variables is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
file-variable =variable-access
```

6.5 Buffer variables

Description

Most file variables have an associated buffer variable, which provides direct access to the next value waiting to be read from the file, or the next value to be written to the file.

In Standard Pascal (i.e. ISO/IEC 7185), file variables can be opened in one of two modes:

1. Inspection - read-only mode (see reset)
2. Generation - write-only mode (see rewrite)

When a file variable is opened in read-only mode, the contents of the buffer variable associated with the file variable, will either be equal to the first value in the file or be undefined if the file is empty. As the file is read, the contents of the buffer variable will always either be equal to the next value to be read from the file, or be undefined if there is no next value to be read from the file (i.e. the entire file has been read). Standard Pascal allows an implementation to delay the point at which the next value to be read from the file, is actually transferred into the buffer variable, until the buffer variable is referenced by the program. This process is a part of what is called *lazy I/O*, where file output operations take place as soon as possible, but file input operations are delayed. *Lazy I/O* is convenient for interactive programs, where it is important that output operations, displaying messages prompting the user to input values, take place before input operations that actually try to read values from the user (so that the user knows what he/she is expected to type in). Irie Pascal implements *lazy I/O*.

When a file variable is opened in write-only mode, the contents of the buffer variable associated with the file variable is initially undefined. You can write values to the file associated with the file variable, by assigning values to the buffer variable associated with the file variable, and using the built-in procedure put to write the contents of the buffer variable to the file.

Irie Pascal allows file variables to be opened in read/write mode, where values can be both read from, and written to the files associated with the file variables. If you reference the buffer variable of a file variable opened in read/write mode, it is unclear whether you intend to read from the file (in which case the next available value from the file should be transferred to the buffer variable), or whether you intend to write to the file (in which case no I/O should take place). Because of this uncertainty it is not recommended that you use buffer variables and the low-level I/O procedures, get and put, on file variable opened in read/write mode. Instead it is recommended that you use the high-level I/O procedures read, write, etc to read and write from file variables opened in read/write mode.

Buffer variables are created and destroyed when their associated file variables are created and destroyed.

Example

The simple example program below uses a file variable (**f**) to create a file called **numbers.dat** and uses the associated buffer variable (**f**) to write the numbers 1 to 10 to the file.

```
program numbers;
var
f : file of integer;
i : integer;
begin
assign(f, 'numbers.dat');
rewrite(f);
for i := 1 to 10 do
begin
f^ := i;
put(f)
end
end.
```

Syntax

The syntax for referencing buffer variables is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
buffer-variable =file-variable '^'|file-variable '@'
```

Where **file-variable** is a reference to a file variable.

6.7 List variables

Description

List variables are used to store values of list types (i.e. collections of values of the same type). The type of each value in a list is called the list's *component type*. An list's *component type* may be any type, including a list type, so it is possible to define lists of lists.

The built-in procedure new must be used to initialize list variables before they can be used to store values, and the built-in procedure dispose should be used to clean-up after list variables that are no longer required.

The number of values in each list is theorectically unlimited (in practice the amount of memory available will limit the size of lists). Each value in a list is identified by index value of integral type. Each value in a list is stored in a seperate component of the list called a list element (or a selected variable). Each element of a list can be referenced individually, in order to store values into, and retrieve values from specific elements. See the syntax section below for details.

Selected Variables

Selected variables are created by calling the built-in procedure insert to insert values into lists, and selected variables are destroyed by calling the built-in procedure delete to remove values from lists. **NOTE:** The built-in procedure dispose will remove all values from lists.

Selected variables are referenced by indexing the list variable that were used to create them.

Example

Here is a simple example program that generates some random numbers, stores them in a list, sorts the numbers, and then prints the numbers out.

```
program sort(output);
const
max = 20;
var
numbers : list of integer;

procedure GenerateRandomNumbers;
var
i : 1..max;
begin
for i := 1 to max do
insert(random(100)+1, numbers);
end;

procedure SortNumbers;
var
i : 1..max;
swapped : boolean;
temp : integer;
begin
repeat
swapped := false;
for i := 1 to max-1 do
if numbers[i] > numbers[i+1] then
begin
temp := numbers[i];
numbers[i] := numbers[i+1];
numbers[i+1] := temp;
swapped := true;
end;
until not swapped;
end;

procedure PrintNumbers;
var
i : 1..max;
begin
for i := 1 to max do
writeln(i, numbers[i]);
end;

begin
randomize;
new(numbers);
GenerateRandomNumbers;
SortNumbers;
PrintNumbers;
dispose(numbers);
end.
```

Syntax

The syntax for referencing selected variables is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
selected-variable =list-variable '['index-expression {','index-expression }']'

index-expression =expression
```

where **list-variable** is a reference to a list variable and **index-expression** is an expression, of integral type, that evaluates to the position of a selected variable in the list.

6.8 Object variables

Description

Object variables are used to store values of object types, each of which either identifies an instance of an object described by the object type or is the built-in constant nil which is guaranteed **not** to reference any instance of an object.

Currently the only supported object types are: the built-in object types connection and recordset, and the generic object type.

Object instances are called instance variables and are described in more detail in the next section.

Instance Variables

Instance variables contain the function and procedure methods, and properties defined by their object type. Function and procedure methods define the *operations* that can be perfomed on and by instance variables, and can be called anywhere a normal function or procedure can be called. Properties define the data (or at least the visible data) contained in the instance variables, and be referenced anywhere normal variables can be referenced.

In the case of object variables of the built-in object types connection and recordset, instance variables are created by calling the built-in procedure new, and passing the object variable as a parameter.

In the case of object variables of the generic object type, instance variables are created by calling the built-in function createobject, and passing the name of a COM object as a parameter.

In all cases, instance variables are destroyed by calling the built-in procedure dispose, and passing the object variable, that was used to create the instance variable, as a parameter.

Syntax

The syntax of referencing function and procedure methods, and properties is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
function-method-designator =object-variable '.'function-method-identifier [actual-
parameter-list ]
```

```
procedure-method-statement =procedure-method-specifier [actual-parameter-list ]

property-designator =object-variable '.'property-specifier

actual-parameter =expression |variable-access |
procedure-identifier |function-identifier

actual-parameter-list ='('actual-parameter {','actual-parameter }')'

function-method-identifier =identifier

object-variable =variable-access

procedure-method-identifier =identifier

procedure-method-specifier =object-variable '.'procedure-method-identifier

property-identifier =identifier

property-specifier =property-identifier |'('property-string ')'

property-string =string-expression
```

where **object-variable** is a reference to an object variable and **property-string** is an expression, of string type, that evaluates to the name of the property being referenced.

6.9 Pointer variables

Description

Pointer variables are used to store values of pointer types. These values, either identify a variable created by using the built-in procedure new on a pointer variable, or are the built-in value nil which is guaranteed **not** to identify any variable created using new.

Variables created by using the built-in procedure new on pointer variables are called identified variables, and are described in more detail in the next section.

Identified Variables

Identified variables are created by calling the built-in procedure new and passing a pointer variable as a parameter. Identified variables are destroyed by calling the built-in procedure dispose and passing, as a parameter, the same pointer variables that were used to create them. You should destroy identified variables after you have finished using them.

Syntax

The syntax for accessing an identified variable is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
identified-variable =pointer-variable '^'|pointer-variable '@'

pointer-variable =variable-access
```

where **pointer-variable** is a reference to a pointer variable.

6.10 Record variables

Description

Record variables are used to store values of record types (i.e. collections of values of possibly different types). Each member of a record type's collection is called a field, and is usually identified by a *name* (i.e. the field name). Fields names have the same syntax as identifiers, and no two fields in a record type can have the same *name*. Every field in a record must have a *name*, except for the variant selector of a variant record for which a *name* is optional.

A reference to a record variable is a reference to the entire record as a unit, and can be used to both store record values into, and retreive record values from the record variable.

Each field in a record is stored in a seperate component of the record, and can be referenced individually, in order to store values into, and retrieve values from specific fields. A reference to an individual field in a record is called a *field designator*. *Field designators* usually consist of a reference to the record variable followed by a period (.), and then finally followed by a *field specifier* (which is what the *name* of the field is called when used in this way). Within the *statement part* of a with statement, *field designators* can consist of just a *field designator identifier* (which is what the *name* of the field is called when used in this way). The record variable is referenced once in the *record variable list* part of the with statement. This can be useful when the *statement part* of a with statement contains many references to the fields of a record.

Syntax

The syntax for field designators is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
field-designator =record-variable '.'field-specifier |field-designator-identifier

field-designator-identifier =identifier

field-identifier =identifier

field-specifier =field-identifier

record-variable =variable-access
```

where **record-variable** is a reference to a record variable, and **field-specifier** and **field-designator-identifier** are field names. **NOTE: field-designator-identifier** can be used only in the *statement part* of a with statement.

6.11 Set variables

Description

Set variables are used to store values of set types.

Example

The following simple example program shows a typical use of set variables to classify values. The program operates as follows:

First the set variable **letters** is initialized with the set of characters containing all lowercase and uppercase characters. Next the set variable **digits** is initialized with the set of characters containing all the digits. Next the user is prompted to enter a character, and this character is read in. Finally the character is classified by checking whether it is a member of **letters**, **digits** or neither. **NOTE:** Sets are often used like this as an alternative to using a big case statement.

```
program classify(input, output);
var
letters, digits : set of char;
c : char;
begin
letters := ['a'..'z', 'A'..'Z'];
digits := ['0'..'9'];
write('Enter a character: ');
read(c);
if c in letters then
writeln('You entered a letter')
else if c in digits then
writeln('You entered a digit')
else
writeln('You entered a symbol')
end.
```

6.12 String variables

Description

String variables are used to store values of string types (i.e. variable length sequences of characters). The maximum number of characters allowed in each sequence is fixed and is defined when the string type is defined. The individual characters in a string can be referenced using a similiar syntax to the that used to reference individual array elements.

Example

For example if **name** is a string variable containing the string 'Bob' then

```
name[1] accesses the first character in the string (i.e. 'B')
```

and

```
name[2] accesses the second character in the string (i.e. 'o')
```

The + operator can be used to perform string concatenation (i.e. joining).

For example here is a hello world program using string concatenation.

```
program good(output);
begin
writeln('hello' + ' ' + 'world' + '!')
end.
```

Syntax

The syntax for accessing individual elements in a string is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
indexed-variable-string =string-variable '['integral-expression ']'

integral-expression =expression

string-variable =variable-access
```

where **string-variable** is a reference to a string variable, and **integral-expression** is an expression which evaluates to the position of the string element being accessed.

6.13.1 The built-in variable identifiers

Built-in variable identifiers are automatically declared by the compiler. Unlike reserved words, you can declare your own identifiers with the same *spelling* as built-in variable identifiers, if you do then your declaration will override the declaration made by the compiler. The built-in variable identifiers are listed below:

- errors
- exitcode
- input
- null
- output

6.13.2 errors

Description

errors is the variable identifier for a built-in list variable which stores the list of most recent trappable errors that were detected. **NOTE:** This list variable is automatically cleared before each operation that could cause a trappable error, so don't expect the list of errors to be maintained indefinately. The operations that can cause trappable errors are **file** and **database** operations.

Portability

Operating Systems: All
Standard Pascal: No

6.13.3 exitcode

Description

exitcode is the variable identifier for a built-in variable of type integer, whose value is returned to the calling program when your program exits.

Example

For example, suppose you want to return a value of 1 from your program (perhaps to indicate that your program detected an error) then you can use the following code

```
exitcode := 1;
halt;
```

NOTE: You can treat **exitcode** like any other integer variable (i.e. you can assign integer values to it, use it in an expression, pass it as an argument to a function, etc).

Portability

Operating Systems: All
Standard Pascal: No

6.13.4 input

Description

input is the variable identifier for a built-in file variable associated with the standard input stream. The standard input stream is usually connected to the computer keyboard.

Portability

Operating Systems: All
Standard Pascal: Yes

6.13.5 null

Description

null is the variable identifier for a special built-in variable of type address, which is implemented by Irie Pascal to make it easier to call certain external functions and procedures written in the C programming language.

In C, there is no special mechanism for passing variables by reference, instead the programmer must explicitly declare the formal parameter to be the address of a variable, and pass the address of a variable as the actual parameter when making the call.

In Pascal, there is a special mechanism (i.e. var parameters) for passing parameters by reference, the programmer would declare the formal parameter as a var parameter, but the call remains the same because the Pascal implementation will take take of the passing the address of the variable behind the scenes.

So when declaring in Pascal, an external function or procedure written in C, that passes variables by reference it is natural to use var parameters for the variables passed by reference. In most cases, this works great and Irie Pascal will take care of the details, however in the following special case there is a problem. Neither C not Pascal support optional parameters (at least not in the standard versions of these languages), however in C because addresses are being passed explicity, and an address is just a value, it is possible to pass special values to mean that the parameter was not passed. In C, the special value zero (called the null pointer) is usually used for this purpose. The problem is, if a parameter is declared as a var parameter in Pascal you can't pass a value like zero, and any variable you do pass will have a non-zero address. The solution is to pass the special variable **null** which has an address of zero.

Operating Systems: All
Standard Pascal: No

6.13.6 output

Description

output is the variable identifier for a built-in file variable associated with the standard output stream. The standard output stream is usually connected to the screen.

Portability

Operating Systems: All
Standard Pascal: Yes

6.14.1 Variable Declarations

Description

Variable identifiers are identifiers that have been associated with a variable using a variable declaration. Variable declarations must be placed in variable declaration groups.

Example

Here is an example of a variable declaration group:

```
var
name : string;
age, heigth : integer;
```

Syntax

The syntax for variable declaration groups is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
variable-declaration-group ='var'variable-declaration {';'variable-declaration }

identifier =letter {letter |digit }

identifier-list =identifier {','identifier }
```

```
new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

type-denoter =type-identifier |new-type

type-identifier =identifier

variable-declaration =identifier-list ':'type-denoter
```

7.1 What are functions and procedures?

Description

Pascal encourages modular programming, where a program is created from a number of smaller and simpler pieces known as modules. Modules can make call other modules (i.e. execute other modules), or even call themselves. Pascal supports two kinds of modules, which are called functions and procedures.

Functions are called in expressions, and each function call has a value in an expression. Procedures on the other hand are called in procedure statements. In all other respects functions and procedures are identical, and the generic term subroutine will be used to refer to both types of modules.

Sometimes it is useful to create subroutines that accept input data and/or produce output data. One way to achieve this to have the subroutines use global variables (i.e. variables declared in the main program block) to store the input and output data. However the overuse of global variables can make programs difficult to understand and change. The basic problem is that since gloval variables are not specific to any particular subroutine, it can be difficult to keep track of which subroutines use which global variables and when. As a result, subroutines can become tightly intertwined making it very difficult to change one subroutine without changing the others.

Another way to create subroutines that accept input data and/or generate output data is to give each subroutine it's own set of special variables to accept input data and store output data. These special variables are called parameters. When a subroutine is declared a possibly empty list of parameters (called formal parameters) is specified, and within the subroutine the formal parameters are referenced to accept input data and store output data. When a subroutine is called the caller must supply the actual data (called actual parameters) for each formal parameter specified by the subroutine declaration.

Pascal supports four kinds of formal parameters (value parameters, variable parameters, function parameters, and procedure parameters).

Value parameters are passed by value (i.e. when subroutines with value parameters are called, the corresponding actual parameters are expressions, and the values of these expressions become the initial values of the value parameters. Subroutines reference value parameters just like normal variables, and can retrieve and change the value stored in them, however changes made to the value stored in value parameters are invisible to the caller.

Variable parameters are passed by reference (i.e. when subroutines with variable parameters are called, the corresponding actual parameters are variables, and the addresses of these variables are passed into the variable parameters). Subroutines reference variable parameters just like normal variables, however references to variable parameters actually reference the variables passed as the corresponding actual parameters.

Function parameters are passed by reference (i.e. when subroutines with function parameters are called, the corresponding actual parameters are functions, and the addresses of these functions are passed into the function parameters. Subroutines can call function parameters just like normal functions, however calling function parameters actually calls the functions passed as the corresponding actual parameters.

Procedure parameters are passed by reference (i.e. when subroutines with procedure parameters are called, the corresponding actual parameters are procedure, and the addresses of these procedure are passed into the procedure parameters. Subroutines can call procedure parameters just like normal procedure, however calling procedure parameters actually calls the procedure passed as the corresponding actual parameters.

Example

For example here is an example of a simple program with a procedure.

```
program simple(output);

procedure say (message : string);
begin
writeln(message)
end;

begin
say('Hello world');
say('Good bye!');
end.
```

The program contains a procedure called **say** which takes a single value parameter. When the procedure is called for the first time the actual parameter is the expression **'Hello world'**, which is passed into the formal parameter **message**. So the initial value of **message** is **'Hello world'**. The call to the built-in procedure writeln causes the value of message (i.e. **'Hello world'**) to be written to the standard output stream. When the procedure is called for the second time the actual parameter is the expression **'Good bye!'**, which is passed into the formal parameter **message**. So the initial value of **message** is **'Good bye!'**. The call to the built-in procedure writeln causes the value of message (i.e. **'Good bye!'**) to be written to the standard output stream.

Pascal supports recursive function and procedure calls (i.e. Pascal allows functions and procedures to call themselves either directly or indirectly). For example a procedure A can call itself directly or it can call a procedure B which in turn calls procedure A.

Functions and procedures must be declared before they can be called. This creates a problem when you have functions or procedures that call each other. For example suppose you have the program below.

```
program test(output);

procedure a(x : integer);
begin
writeln(x);
b(x+1)
end;

procedure b(x : integer);
begin
```

```
writeln(x);
a(x+1)
end;

begin
a(1);
end.
```

If you try to compile this program, the compiler will complain about the call **b(x+1)** that occurs in **procedure a** because this call occurs before **procedure b** is declared. You can try to correct this problem by moving the declaration of **procedure b** before the declaration of **procedure a** like so

```
program test(output);

procedure b(x : integer);
begin
writeln(x);
a(x+1)
end;

procedure a(x : integer);
begin
writeln(x);
b(x+1)
end;

begin
a(1);
end.
```

but if you try to compile this program, the the compiler will now complain about the call to **a(x+1)** that occurs in **procedure b**, because this call now occurs before **procedure a** is declared.

The solution to this problem is to use the forward directive as illustrated by the program below:

```
program test(output);

procedure b(x : integer); forward;

procedure a(x : integer);
begin
writeln(x);
b(x+1)
end;

procedure b;
begin
writeln(x);
a(x+1)
end;

begin
a(1);
end.
```

See the forward directive for more information.

As an extension to Standard Pascal, Irie Pascal supports calling functions and procedures inside Windows DLLs using the external directive.

See the external directive for more information.

Syntax

The syntax for function and procedure declarations is given below:

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
function-declaration =
function-heading ';'directive |
function-identification ';'function-block |
function-heading ';'function-block

procedure-declaration =
procedure-heading ';'directive |
procedure-identification ';'procedure-block |
procedure-heading ';'procedure-block

block =declarative-part statement-part

compound-statement ='begin'statement-sequence 'end'

declaration-group =
label-declaration-group |
constant-definition-group |
type-definition-group |
variable-declaration-group |
function-declaration |
procedure-declaration

declarative-part ={declaration-group }

directive =forward-directive |external-directive

formal-parameter-list ='('formal-parameter-section {';'formal-parameter-section }')'

formal-parameter-section =value-parameter-specification |
variable-parameter-specification |
procedure-parameter-specification |
function-parameter-specification

function-block =block

function-heading ='function'identifier [formal-parameter-list ]':'result-type

function-identification ='function'function-identifier

function-identifier =identifier

function-parameter-specification =function-heading

identifier =letter {letter |digit }

identifier-list =identifier {','identifier }

procedure-block =block

procedure-heading ='procedure'identifier [formal-parameter-list ]

procedure-identification ='procedure'procedure-identifier

procedure-identifier =identifier

procedure-parameter-specification =procedure-heading

result-type =type-identifier

statement-part =compound-statement
```

```
statement-sequence =statement {';'statement }

type-identifier =identifier

value-parameter-specification =identifier-list ':'type-identifier

variable-parameter-specification ='var'identifier-list ':'type-identifier
```

7.2.1 The built-in functions

The **built-in functions** are functions which are automatically declared by the compiler, and each **built-in function** is associated with a built-in function identifier. Unlike reserved words, you can declare your own identifiers with the same spelling as a built-in function identifier, and if you do so then your declaration will override the declaration made by the compiler. The built-in function identifiers are listed below:

- abs
- addr
- arctan
- chr
- concat
- copy
- copyword
- cos
- cosh
- countwords
- createobject
- dirsep
- eof
- eoln
- exp
- fexpand
- filematch
- filepos
- filesize
- frac
- getenv
- getlasterror
- hex
- int
- ioresult
- isalpha
- isalphanum
- isdigit
- islower
- isnull
- isprint
- isspace
- isupper
- isxdigit
- keypressed
- length
- ln
- log
- locase
- lowercase

7.2.2 The abs function

Description

The **abs** function returns the absolute value (i.e. the magnitude) of the parameter passed to it. The type of the value returned by this function is always the same as the type of it's parameter.

Parameter

The **abs** function's only parameter is the numeric expression (i.e. an expression of integral type or of real type) whose absolute value is to be returned.

Example

```
abs(-10)   returns   10
abs(10)   returns  10
abs(-6.7) returns  6.7
abs(6.7)  returns  6.7
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.3 The addr function

Description

The **addr** function returns the operating system address of the parameter passed to it. The type of the value returned by this function is address. This function is most often used when calling external functions and procedures.

Parameter

The **addr** function's only parameter is a reference to the variable whose address is to be returned.

Portability

Operating Systems: All
Standard Pascal: No

7.2.4 The arctan function

Description

The **arctan** function returns the arctangent in radians of the parameter passed to it. So if **X** is the parameter passed to this function, then the value returned by this function **Y** is the angle in radians satisfying the following equation:

`X = tan(Y)`

The type of the value returned by this function is always real.

Parameter

The **arctan** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type).

Example

arctan(0.5) returns 4.64E-1

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

7.2.5 The chr function

Description

The **chr** function returns the character whose ordinal value is equal to the parameter passed to it. So if **X** is the parameter passed to this function, then the value returned by this function **Y** is the character value satisfying the following equation:

```
X = ord(Y)
```

NOTE: It is an error if there is no character whose ordinal value is equal to the parameter.

Parameter

The **chr** function's only parameter is an expression of integral type.

Example

chr(64) returns '@' (assuming the ASCII or ANSI character set).

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

7.2.6 The concat function

Description

The **concat** function returns a string value formed by concatenating (i.e. joining) the parameters passed to it.

Parameters

The **concat** function's parameters are one or more expressions of string type or char type.

Example

concat('abc', '123', '!') returns 'abc123!'

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.7 The copy function

Description

The **copy** function accepts up to three of the parameters described below (the last parameter is optional), and returns a value of type string which is formed by copying zero or more characters from the first parameter.

Parameters

1. The first parameter **S** is an expression of type string or type char, and contains the characters to be copied.
2. The second parameter **P** is an expression of integral type, and specifies the position, in **S** of the first character to copy. If **P** is greater than the number of charaters in **S** then no characters are copied.
3. The third parameter **N** is an expression of integral type, and specifies the number of characters to copy. If **N** is omitted or is greater than the number of characters in **S** from **P** to the end, then all characters in **S** starting from **P** to the end are returned.

Example

```
copy('Testing...', 5, 2)         returns   'in'
copy('Testing...', 7)             returns   'g...'
copy('Testing...', 1, 1000)    returns   'Testing...'
copy('Testing...', 50)            returns   ''
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.8 The copyword function

Description

The **copyword** function accepts up to three of the parameters described below (the last parameter is optional), and returns a value of type string which is formed by copying a *word* from the first parameter.
NOTE: In this help topic the term *word* is used to mean a group of contiguous characters, seperated by another group of characters called deliminators.

Parameters

1. The first parameter (**S**) is an expression of type string or type char, and contains the *words* to be copied.
2. The second parameter (**I**) is an expression of integral type, and specifies the *word* to copy. *Words* are numbered starting from one, so it is an error if **I** is less than one. If **I** is greater than the number of *words* in **S** then no *word* is copied.
3. The third parameter (**D**) is an expression of type string or type char, and contains the delimintors used to seperate *words*. If **D** is not present then the default deliminators (i.e. SPACE, TAB, RETURN, and LINEFEED) are used.

Example

```
copyword('example', 1)                    returns  'example'
copyword('   example    ', 1)             returns  'example'
copyword('this is an example', 2)      returns  'is'
copyword('example', 2)                    returns  ''
```

Here are some examples using "," as the deliminstor, to extract fields from comma-deliminated strings.

```
copyword('1234, name, value', 1, ',')  returns  '1234'
copyword('1234, name, value', 2, ',')  returns  ' name'
copyword('1234, name, value', 3, ',')  returns  ' value'
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.9 The cos function

Description

The **cos** function returns the cosine of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **cos** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type) which is an angle in radians.

Example

```
cos(pi/3)  returns  0.5
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.10 The cosh function

Description

The **cosh** function returns the hyperbolic cosine of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **cosh** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type) which is an angle in radians.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.11 The countwords function

Description

The **countwords** function accepts up to two of the parameters described below (the last parameter is optional), and returns a value of type integer type, which is equal to the number of *words* in the first parameter. The term *word* is used to mean a group of contiguous characters, seperated by another group of characters called deliminators.

Parameters

1. The first parameter (**S**) is an expression of type string or type char, and contains the *words* to be counted.
2. The second parameter (**D**) is an expression of type string or type char, and contains the delimintors used to seperate *words*. If **D** is not present then the default deliminators (i.e. SPACE, TAB, RETURN, and LINEFEED) are used.

Example

```
countwords('example')             returns  1
countwords('this is an example')  returns  4
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.12 The createobject function

Description

The **createobject** function creates an instance variable of the generic object type (i.e. an instance of the generic object type), and returns a value that identifies the instance variable.

Parameter

The **createobject** function's only parameter is an expression of string type or char type which is the name of the COM object class to create an instance of.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.13 The dirsep function

Description

The **dirsep** function returns a value of type string that is equal to the directory separator character for the current platform. **NOTE:** On Windows the value returned is "\", and on Linux, FreeBSD, Solaris/x86 and Solaris_Sparc the value returned is "/".

Parameters

None.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.14 The eof function

Description

The **eof** function tests whether a file is at end-of-file, and returns a value of boolean type that indicates the result of the test. If the file tested is at end-of-file then the value true is returned, and if the file tested in not at end-of-file then the value false is returned.

Parameter

The **eof** function's optional parameter is a reference to a file variable. If the optional parameter is supplied then the **eof** function tests the file associated with the parameter. If the optional parameter is not supplied then the file associated with the built-in variable input is tested.

Example

The function below counts the number of lines in a text file. The **eof** function is used to determine when the end of the text file has been reached.

```
function CountLines(name : filename) : integer;
var
count : integer;
f : text;
begin
count := 0;
reset(f, name);
while not eof(f) do
begin
readln(f);
inc(count);
end;
CountLines := count;
close(f)
end;
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.15 The eoln function

Description

The **eoln** function tests whether a text file is at end-of-line, and returns a value of type boolean that indicates the result of the test. If the file tested is at end-of-line then the value true is returned, and if the file tested in not at end-of-file then the value false is returned.

Parameter

The **eoln** function's optional parameter is a reference to a file variable of type text. If the optional parameter is supplied then the **eoln** function tests the file associated with the parameter. If the optional parameter is not supplied then the file associated with the built-in variable input is tested.

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.16 The exp function

Description

The **exp** function returns the value of the mathematical constant **e** raised to a power. The type of the value returned by this function is always real.

Parameter

The **exp** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type) which specifies the power to which **e** should be raised to.

Example

```
exp(2)   returns   7.389...
```

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

7.2.17 The fexpand function

Description

The **fexpand** function expands a filename into a full pathname, and returns the full pathname. The type of the value returned by this function is always string.

Parameter

The **fexpand** function's only parameter is an expression of string type or char type which contains the filename to to be expanded. **NOTE:** If the filename is already expanded then it is returned unchanged.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.18 The filematch function

Description

The **filematch** function accepts two parameters as described below, and returns a value of type boolean, which indicates whether the filename matches the file specification. If the filename matches the file specification then the value true is returned, and if the filename does not match the file specification then the value false is returned.

Parameter

1. The first parameter (**file-spec**) is an expression of type string or type char, which contains the file specification with which the filename is being matched. The file specification may contain the following wildcard characters('*' and '?'). The character '*' if used in the file specification matches any zero or more characters in the filename. The character '?' if used in the file specification matches any single character in the filename.
2. The second parameter (**filename**) is an expression of type string or type char, which contains the filename to be matched with the file specification. The filename is not tested for validity nor does the file have to exist.

Example

```
filematch('file.txt', 'file.txt')  returns  true
filematch('file.txt', 'otherfile.txt')  returns  false
filematch('*.txt', 'file.txt')  returns  true
filematch('*.txt', 'otherfile.txt')  returns  true
filematch('a?c', 'abc')  returns true
filematch('a?c', 'agc')  returns true
filematch('a?c', 'azc')  returns true
filematch('a?', 'abc')  returns false
filematch('a?c', 'ab')  returns false
filematch('?c', 'bc')  returns false
filematch('??c', 'abc')  returns true
filematch('a??', 'avd')  returns true
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.19 The filepos function

Description

The **filepos** function returns the current file position of a file. The type of the value returned by this function is always integer.

Parameter

The **filepos** function's only parameter is a reference to the file variable, that specifies the file whose current file position is to be returned. **NOTE:** For a file variable of type text the value of the file position may not be equal to the number of characters read/written to the file so far. **NOTE:** For file variable of other types, the value of the file position is equal to the number of characters read/written so far. The file variable must refer to an open file.

Portability

Operating Systems: All
Standard Pascal: No

7.2.20 The filesize function

Description

The **filesize** function returns the size (in bytes) of a file. The type of the value returned by this function is always integer.

Parameter

The **filesize** function's only parameter is a reference to the file variable that specifies the file whose size is to be returned. **NOTE:** The file variable must refer to an open file.

Portability

Operating Systems: All
Standard Pascal: No

7.2.21 The frac function

Description

The **frac** function returns the fractional part of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **frac** function's only parameter is the expression of real type whose fractional part is to be returned.

Example

```
frac(7.234)  returns 0.234
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.22 The getenv function

Description

The **getenv** function returns the value of an environment variable. The type of the value returned by this function is always string.

Parameter

The **getenv** function's only parameter is the expression of string type or char type which contains the name of the environment variable whose value is to be returned. **NOTE:** If no environment variable has a name matching this parameter then an empty string is returned.

Portability

Operating Systems: All
Standard Pascal: No

7.2.23 The getlasterror function

Description

The **getlasterror** function returns the number of the last trappable error that occured. The type of the value returned by this function is always integer.

When error trapping is enabled your program automatically checks for trappable errors after each operation that could cause a trappable error. Your program will terminate if a trappable error occurs. If you do not want this default behavior you can disable error trapping using the built-in procedure traperrors and call the **getlasterror** function to determine if any errors occurred.

After this function is called the error number is cleared to zero (this prevents the function from returning the same error more than once). However because of this, if you need to refer to the error number more than once you should store the result of the first call in a variable.

New programs should use **getlasterror** instead of ioresult, which dispite it's name also returns all trappable errors not just I/O errors. ioresult continues to be supported for compatibility reasons only.

Parameter

None.

Example

For example the following is incorrect:

```
if getlasterror <> 0 then
case getlasterror of
```

```
1: writeln('Error erasing file');
2: writeln('Error renaming file');
3: writeln('File is undefined')
end
```

since the second call to **getlasterror** will always return 0. Instead you should do something like this:

```
errnum := getlasterror;
if errnum <> 0 then
case errnum of
1: writeln('Error erasing file');
2: writeln('Error renaming file');
3: writeln('File is undefined')
end
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.24 The hex function

Description

The **hex** function returns the hexadecimal value of the parameter passed to it. The type of the value returned by this function is always string.

Parameter

The **hex** function's only parameter is the expression of integral type whose value is to be converted to hexadecimal.

Example

```
hex(10)   returns  'A'
hex(16)   returns  '10'
hex(255)  returns  'FF'
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.25 The int function

Description

The **int** function returns the integral part of the parameter passed to it. The type of the value returned by this function is always real. **NOTE:** The return type is real instead of one of the integral types because the integral part of the parameter may be too big to fit in any of the integral types, and you can always use the built-in function trunc to convert the return value from real to integer.

Parameter

The **int** function's only parameter is the expression of real type whose integral part is to be returned.

Example

```
int(7.234)   returns  7.0
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.26 The ioresult function

Description

The **ioresult** function returns the number of the last trappable error that occured. Use of this function is no longer recommended, new programs should use getlasterror instead.

Parameter

None.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.27 The isalpha function

Description

The **isalpha** function returns a value of type boolean, which indicates whether the parameter passed to it is an alphabetic character. If the parameter is an alphabetic character then the value true is returned, and if the parameter is not an alphabetic character then the value false is returned.

Parameter

The **isalpha** function's only parameter is an expression of char type.

Example

```
isalpha('a')  returns   true
isalpha('A')  returns   true
isalpha('1')  returns   false
isalpha('@')  returns   false
isalpha('_')  returns   false
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.28 The isalphanum function

Description

The **isalphanum** function returns a value of type boolean, which indicates whether the parameter passed to it is an alphanumeric character (i.e. an alphabetic character or a digit). If the parameter is an alphanumeric character then the value true is returned, and if the parameter is not an alphanumeric character then the value false is returned.

Parameter

The **isalphanum** function's only parameter is an expression of char type.

Example

```
isalphanum('a')  returns   true
isalphanum('A')  returns   true
isalphanum('1')  returns   true
isalphanum('@')  returns   false
isalphanum('_')  returns   false
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.29 The isdigit function

Description

The **isdigit** function returns a value of type boolean, which indicates whether the parameter passed to it is a digit. If the parameter is a digit then the value true is returned, and if the parameter is not a digit then the value false is returned.

Parameter

The **isdigit** function's only parameter is an expression of char type.

Example

```
isdigit('a')  returns   false
isdigit('A')  returns   false
isdigit('1')  returns   true
isdigit('@')  returns   false
isdigit('_')  returns   false
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.30 The islower function

Description

The **islower** function returns a value of type boolean, which indicates whether the parameter passed to it is a lowercase alphabetic character. If the parameter is a lowercase alphabetic character then the value true is returned, and if the parameter is not a lowercase alphabetic character then the value false is returned.

Parameter

The **islower** function's only parameter is an expression of char type.

Example

```
islower('a')  returns   true
islower('A')  returns   false
islower('1')  returns   false
islower('@')  returns   false
islower('_')  returns   false
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.31 The isnull function

Description

The **isnull** function returns a value of type boolean, which indicates whether the parameter passed to it is null. If the parameter is null then the value true is returned, and if the parameter is not null then the value false is returned. This function is usually used to test values retrieved from databases with recordset.field.

Parameter

The **isnull** function's only parameter is an expression of variant type.

Example

The following code fragment uses **isnull** to check the values of the **last_name** and **first_name** fields to see if they are null, and if so handle them specially.

```
new(rs);
rs.open(conn, 'select last_name, first_name from customer', rsforward);
while not rs.eof
begin
if isnull(rs.field('last_name')) then
write('NULL')
else
write(rs.field('last_name'));
write(', ');
if isnull(rs.field('first_name')) then
writeln('NULL')
else
writeln(rs.field('first_name'))
rs.movenext;
end;
rs.close;
dispose(rs);
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.32 The isprint function

Description

The **isprint** function returns a value of type boolean, which indicates whether the parameter passed to it is a printable character. If the parameter is a printable character then the value true is returned, and if the parameter is not a printable character then the value false is returned.

Parameter

The **isprint** function's only parameter is an expression of char type.

Example

```
isprint('A')     returns   true
isprint('3')     returns   true
isprint('+')     returns   true
isprint(' ')     returns   true
isprint(chr(10)) returns   false
isprint(chr(8)) returns    false
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.33 The isspace function

Description

The **isspace** function returns a value of type boolean, which indicates whether the parameter passed to it is a whitespace character (i.e. space, tab, linefeed, formfeed, or carriage return). If the parameter is a whitespace character then the value true is returned, and if the parameter is not a whitespace character then the value false is returned.

Parameter

The **isspace** function's only parameter is an expression of char type.

Example

```
isspace(' ')     returns   true
isspace(chr(9))  returns   true  (assuming chr(9) is TAB)
isspace(chr(10)) returns   true  (assuming chr(10) is LINEFEED)
isspace(chr(13)) returns   true  (assuming chr(13) is CARRIAGE RETURN)
isspace('A')     returns   false
isspace('6')     returns   false
isspace('$')     returns   false
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.34 The isupper function

Description

The **isupper** function returns a value of type boolean, which indicates whether the parameter passed to it is a uppercase alphabetic character. If the parameter is a uppercase alphabetic character then the value true is returned, and if the parameter is not a uppercase alphabetic character then the value false is returned.

Parameter

The **isupper** function's only parameter is an expression of char type.

Example

```
isupper('a')  returns   false
isupper('A')  returns   true
isupper('1')  returns   false
isupper('@')  returns   false
isupper('_')  returns   false
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.35 The isxdigit function

Description

The **isxdigit** function returns a value of type boolean, which indicates whether the parameter passed to it is a hexadecimal digit character (i.e. '0'..'9', and 'a'..'f', and 'A'..'F'). If the parameter is a hexadecimal digit character then the value true is returned, and if the parameter is not a hexadecimal digit character then the value false is returned.

Parameter

The **isxdigit** function's only parameter is an expression of char type.

Example

```
isxdigit('a')  returns   true
isxdigit('A')  returns   true
isxdigit('h')  returns   false
isxdigit('1')  returns   true
isxdigit('@')  returns   false
isxdigit('_')  returns   false
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.36 The keypressed function

Description

The **keypressed** function returns a value of type boolean, which indicates whether there are keys in the keyboard buffer, waiting to be read. If there are keys in the keyboard buffer then the value true is returned, and if there are no keys in the keyboard buffer then the value false is returned.

Parameter

None.

Portability

Operating Systems: Windows
Standard Pascal: No

7.2.37 The length function

Description

The **length** function returns the number of characters in a string or character, or the number of values in a list. The type of the value returned by this function is always integer.

Parameter

The **length** function's only parameter is an expression of one of the following types:

- string type - The **length** function returns the number of characters in the string.
- char type - The **length** function returns one.
- list type - The **length** function returns the number of values in the list.

Example

```
length('')        returns   0
length('a')       returns  1
length('hello')   returns  5
```

And if **l** is a list of integer then

```
new(l);
length(l);         returns  0
```

and

```
new(l);
insert(1, l);
insert(78, l);
insert(45, l);
insert(3, l);
length(l);          returns   4
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.38 The ln function

Description

The **ln** function returns the natural logarithm of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **ln** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type).

Example

```
ln(1)        returns   0.0
ln(exp(1))   returns   1.0
ln(100)      returns   4.61
```

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

7.2.39 The log function

Description

The **log** function returns the logarithm to the base 10 of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **log** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type).

Example

```
log(1)    returns   0.0
ln(10)    returns   1.0
ln(100)   returns   2.0
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.40 The locase function

Description

The **locase** function returns the lowercase letter corresponding to the parameter passed to it. **NOTE:** If there is no lowercase letter corresponding to the parameter passed to it then the parameter is returned unchanged. The type of the value returned by this function is always char.

Parameter

The **locase** function's only parameter is an expression of char type.

Example

```
locase('a')  returns  'a'
locase('A')  returns  'a'
locase('H')  returns  'h'
locase('1')  returns  '1'
locase('+')  returns  '+'
locase('_')  returns  '_'
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.41 The lowercase function

Description

The **lowercase** function returns a string formed by converting all uppercase characters in the parameter passed to it, into lowercase characters, and leaving all non-uppercase characters in the parameter unchanged.

Parameter

The **lowercase** function's only parameter is an expression of string type or char type.

Example

```
lowercase('Hello!!')  returns   'hello!!'
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.42 The odd function

Description

The **odd** function returns a value of type boolean, which indicates whether the parameter passed to it is odd (i.e. not evenly divisible by two). If the parameter is odd then the value true is returned, and if the parameter is not odd then the value false is returned.

Parameter

The **odd** function's only parameter is an expression of integral type.

Example

```
odd(0)   return   false
odd(1)   return   true
odd(2)   return   false
odd(-43)   return true
odd(-44)   return false
```

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

7.2.43 The ord function

Description

The **ord** function returns the ordinal value of the parameter passed to it. The type of the value returned by this function is always integer.

Parameter

The **ord** function's only parameter is an expression of ordinal type.

Example

```
ord(false)    returns  0
ord(true)     returns  1
ord(21)       returns  21
ord('A')      returns  65 (assuming the ASCII or ANSI character set)
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.44 The paramcount function

Description

The **paramcount** function returns the number of arguments passed to the program. The type of the value returned by this function is always integer.

Parameters

None.

Example

For example, the following is a simple program **args.pas**, that uses the **paramcount** function to display the number of arguments passed to it.

```
program args(output);
begin
writeln('Number of program arguments =', paramcount)
end.
```

If you run the program with four arguments like below

```
ivm args this is a test
```

then the output would be

```
Number of program arguments = 4
```

since four arguments were passed to the program

1. this
2. is
3. a

4. test

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.45 The paramstr function

Description

The **paramstr** function returns one of the arguments passed to the program. The parameter passed to this function is the number of the argument to return. The type of the value returned by this function is always string.

Parameter

The **paramstr** function's only parameter is the expression of integral type whose value is the number of the program argument to return (i.e. if the parameter is **N** then this function will return the **N**th program argument). If there is no **N**th program argument then an empty string is returned. If the value of the parameter is zero then the name of the program is returned. If the value of the parameter is minus 1 (i.e. -1) then the name of the interpreter is returned, unless the program is an .EXE executable in which case an empty string is returned.

Example

For example, the following is a simple program **args2.pas**, that uses the **paramstr**, and paramcount functions to display all of the arguments passed to it.

```
program args2(output);
var
i : integer;
begin
for i := -1 to paramcount do
writeln('Program argument #', i:1, ' is ', '"', paramstr(i), '"')
end.
```

If you run this program as as follow:

```
ivm args2 this is a test
```

then the program will display

```
Program argument #-1 is "ivm"
Program argument #0 is "args2"
Program argument #1 is "this"
Program argument #2 is "is"
Program argument #3 is "a"
Program argument #4 is "test"
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.46 The pi function

Description

The **pi** function returns the value of the mathematical constant pi. The type of the value returned by this function is always real.

Parameter

None.

Example

So for example:

```
writeln(pi)  displays   3.14E+00
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.47 The platform function

Description

The **platform** function returns a value that indicates the operating system platform that the program is running on. **NOTE:** Actually the value returned by this function indicates the edition of Irie Pascal that is executing the program. The type of the value returned by this function is always integer, and is one of the platform constants.

Parameter

None.

Example

For example the program below uses the **platform** function to display the name of the platform it's running on.

```
program ShowPlatform(output);
begin
case platform of
platform_dos: writeln('DOS');
platform_os2: writeln('OS/2');
platform_win32: writeln('Windows');
platform_linux: writeln('Linux');
platform_fbsd: writeln('FreeBSD');
platform_solaris: writeln('Solaris/x86');
platform_solaris_sparc: writeln('Solaris/Sparc');
platform_error: writeln('Unknown Platform');
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.48 The pos function

Description

The **pos** function accepts up to three of the parameters described below (the last parameter is optional), searches for one string in another string, and returns the position of the first string in the second string, or zero if the first string was not found in the second string.

Parameter

1. The first parameter **needle** is an expression of string type or char type, and is the string to search for.
2. The second parameter **haystack** is an expression of string type or char type, and is the string to be searched.
3. The third parameter **start** is an expression of integral type, and specifies the position in **haystack** to begin searching for **needle**. If **start** is omitted then **haystack** is searched starting from position one. If **start** is greater than the length of **haystack** then zero is returned.

Example

```
pos('o', 'Hello world')     returns  5
pos('o', 'Hello world', 1)  returns  5
pos('o', 'Hello world', 6)  returns  8
pos('o', 'Hello world', 9)  returns  0
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.49 The pred function

Description

The **pred** returns the ordinal value that is the predecessor of the parameter passed to it. It is an error if the parameter does not have a predecessor. The type of the value returned by this function is always the same as the The type of the parameter.

Parameter

The **pred** function's only parameter is an expression of ordinal type.

Example

```
pred(true)    returns  false
pred(21)      returns  20
pred('A')     returns  '@' (assuming the ASCII or ANSI character set)
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.50 The ptr function

Description

The **ptr** function converts an operating system address into a virtual machine address. The type of the value returned by this function is always a generic pointer type, which is compatible with all other pointer types.

Parameter

The **ptr** function's only parameter is an expression of integral type, or address type.

Portability

Operating Systems: All
Standard Pascal: No

7.2.51 The random function

Description

The **random** function returns a random number. The type of the value returned by this function is either integer, or real depending on whether the optional parameter is supplied.

Parameter

The **random** function's optional parameter is an expression of integral type. If this parameter is supplied then the value returned by this function is a random integer between zero and the value of the parameter minus 1 (i.e. if the value of the parameter is **I** then the value returned by this function is a random integer between zero and **I-1**). If the parameter is not supplied then the value returned by this function is a random real number between zero and one.

Example

```
random(10);     //returns a random integer between 0 and 9
random;         //returns a random real number between 0 and 1
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.52 The readkey function

Description

The **readkey** function returns the next key in the keyboard buffer. If there are no keys in the keyboard buffer then this function waits until a key is pressed. You can use the keypressed function to check the keyboard buffer for keys before calling this function. The type of the value returned by this function is always char.

If the next key in the keyboard buffer corresponds to a printable character then this function simply returns the character, however if the key does not correspond to a printable character (e.g. **Page Up**) then this function actually returns two characters on separate calls to this function. In other words this function has to be called twice to completly read non-printable keys from the keyboard buffer. The first call to this function returns **chr(0)** and the second call returns the scan code for the key.

Parameter

None.

Portability

```
Operating Systems:Windows
Standard Pascal:No
```

7.2.53 The reverse function

Description

The **reverse** function returns a value of type string that is formed by reversing the order of the characters in the parameter passed to it. The type of the value returned by this function is always string.

Parameter

The **reverse** function's only parameter is an expression of string type or char type.

Example

```
reverse('Hello!!')   returns  '!!olleH'
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.54 The round function

Description

The **round** function rounds the parameter passed to it (i.e. returns the integer number that is closest in value to the parameter). The type of the value returned by this function is always integer.

Parameter

The **round** function's only parameter is an expression of real type.

Example

```
round(7.4)  returns  7
round(7.5)  returns  8
round(-7.4)  returns -7
round(-7.5)  returns -8
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.55 The sin function

Description

The **sin** function returns the sine of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **sin** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type) which is an angle in radians.

Example

```
sin(pi/2)  returns  1.0
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.56 The sinh function

Description

The **sinh** function returns the hyperbolic sine of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **sinh** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type) which is an angle in radians.

Example

```
sinh(1)   returns  1.18E+00
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.57 The sizeof function

Description

The **sizeof** function returns the size (in bytes) of it's parameter. The type of the value returned by this function is always word.

Parameter

The **sizeof** function's only parameter is either a reference to a variable, or a type. If the parameter is a reference to a variable then this function returns the number of bytes occupied by the variable. If the parameter is a type then this function returns the number of bytes occupied by variables of that type.

Example

```
sizeof(integer)  returns   4
sizeof(char)     returns   1
sizeof(real)     returns   8
sizeof(boolean)  returns   4
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.58 The sqr function

Description

The **sqr** function returns the square of the value of the parameter passed to it. The type of the value returned by this function is always the same as the type of the parameter.

Parameter

The **sqr** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type).

Example

```
sqr(1)   returns  1
sqr(3)   returns  9
sqr(-5)  returns 25
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.59 The sqrt function

Description

The **sqrt** function returns the square root of the value of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **sqrt** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type).

Example

```
sqrt(1)   returns  1.0
sqrt(9)   returns  3.0
sqrt(25)  returns  5.0
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.60 The stopserviceevent function

Description

The **stopserviceevent** function returns the handle associated with the StopServiceEvent that is sent to services as a signal to stop running. This function returns zero if the program is not a service. The type of the value returned by this function is always word. This function is usually used with the wait function in services to repond to requests to stop running. See the Irie Pascal User's Manual for more information about writing services.

Parameter

None.

Portability

Operating Systems: All
Standard Pascal: No

7.2.61 The succ function

Description

The **succ** returns the ordinal value that is the successor of the parameter passed to it. It is an error if the parameter does not have a succssor. The type of the value returned by this function is always the same as the type of the parameter.

Parameter

The **succ** function's only parameter is an expression of ordinal type.

Example

```
succ(false)   returns  true
succ(21)      returns  22
succ('A')     returns  'B' (assuming the ASCII or ANSI character set)
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.62 The supported function

Description

The **supported** function returns a value of type boolean, which indicates whether a particular feature is supported on the current platform. The parameter passed to this function identifies the feature to be tested for support. If the feature is supported then the value true is returned, and if the feature is not supported then the value false is returned.

Irie Pascal defines a constant for each feature whose availability can be checked with this function. It is recommended that you use one of these predefined constants (as seen in the example above) to indicate which feature you want to check the availability of. This makes your program easier to read and is probably easier than remembering which values corresponds to which features.

Parameter

The **supported** function's only parameter is an expression of integral type, and must be equal to one of the feature constants.

Example

```
supported(feature_odbc)  returns  trueif the current
platform supports ODBC and
falseotherwise.
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.63 The swap function

Description

The **swap** function returns an integer value that is calculated by reversing the order of the bytes in the parameter passed to it. In other words this function converts little endian integer values to big endian integer values and vice-versa.

Parameter

The **swap** function's only parameter is an expression of one of the following types:

- byte - In this case the **swap** function will convert the single byte parameter into a four byte integer, reverse the order of the four bytes making up this integer, and then return the reversed bytes as a value of type integer.
- integer, or subrange of integer - In this case the **swap** function will reverse the order of the four bytes making up the parameter, and then return the reversed bytes as a value of type integer.
- shortint - In this case the **swap** function will convert the two byte parameter into a four byte integer, reverse the order of the four bytes making up this integer, and then return the reversed bytes as a value of type integer.
- shortword - In this case the **swap** function will convert the two byte parameter into a four byte integer, reverse the order of the four bytes making up this integer, and then return the reversed bytes as a value of type integer.
- word, or subrange of word - In this case the **swap** function will reverse the order of the four bytes making up the parameter, and then return the reversed bytes as a value of type word.

Notes

The **swap** function is designed to swap four byte (i.e. 32-bit) integral values. If you pass two byte (i.e. 16-bit), or one byte (i.e. 8-bit) integral values, they are converted to four byte integral values before the swap is performed. As a result of this conversion the result of the swap might not be what you would expect. See the examples below for more information.

Example

For example, assuming 32 bit little endian integers then 256 is stored in four bytes as follows:

```
0 0 1 0
```

So swap(256) results in the following integer:

```
0 1 0 0
```

which is equal to 65536.

As mentioned in the Notes above swapping two byte or one byte values might not give you the results you expect. So what do you do if you have two byte values that you want to swap, well consider the following program:

```
program example(output);
var
x : shortint;
begin
x:=$1234;
x:=swap(x);
writeln(hex(x));
end.
```

This program does not produce the output that you might expect (i.e. 3412). This is because the **swap** function will convert its parameter into an four byte integer (producing the following bytes $00 $00 $12 $34), and then swap around these bytes (producing $34 $12 $00 $00). Actually a run-time error (vaue out of range) will be issued because the value returned by the **swap** function (i.e. $34120000) is too large to fit in a variable of type shortint. As you can see the value you probably want is stored in the upper 16-bits of the result while the lower 16-bits contain zero. The solution to this problem is to shift the value in the upper 16-bits into the lower 16-bits, before the assignment into the shortint variable. You can do this by changing the

```
x:=swap(x);
```

into either

```
x:= swap(x) shr 16;
```

or

```
x:=swap(x) div $10000;
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.64 The system function

Description

The **system** function passes a command to the command processor for execution, and returns the resulting error code. If there is an error executing the command then a non-zero value indicating the kind of error is returned. If there is no error executing the command then the value returned is zero. The type of the value returned by this function is always integer.

Parameter

The **system** function's only parameter is an expression of string type or char type, and is the command to send to the command processor.

Example

For example the simple program **batch.pas** below is a very primitive batch processor. It sends each line in the input file to the command processor to be executed.

```
program batch(f, output);
var
f : text;
s : string;
err : integer;
begin
reset(f);     (* open input file or standard input file *)
while not eof(f) do
begin
readln(f, s);
writeln('Executing ', s);
err := system(s);      (* Pass 's' to the command processor *)
writeln('Error code is ', err)
end
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.65 The tan function

Description

The **tan** function returns the tangent of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **tan** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type), which is an angle in radians.

Example

```
tan(pi/4)  returns  1.0
```

Portability

Operating Systems: All

Standard Pascal: No

7.2.66 The tanh function

Description

The **tanh** function returns the hyperbolic tangent of the parameter passed to it. The type of the value returned by this function is always real.

Parameter

The **tanh** function's only parameter is a numeric expression (i.e. an expression of integral type or of real type), which is an angle in radians.

Example

```
tanh(1)   returns   0.762
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.67 The trim function

Description

The **trim** function returns a string formed by removing all leading and trailing spaces from the parameter passed to it.

Parameter

The **trim** function's only parameter is an expression of string type or char type.

Example

```
trim(' hello!! ')  returns 'hello!!'
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.68 The trunc function

Description

The **trunc** function truncates the parameter passed to it (i.e. returns the integer number part of the the parameter). The type of the value returned by this function is always integer.

Parameter

The **trunc** function's only parameter is an expression of real type.

Example

```
trunc(7.234)  returns 7
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.2.69 The unixplatform function

Description

The **unixplatform** function returns a value of type boolean, which indicates whether the program is running on a UNIX-like operating system (currently Linux, FreeBSD, Solaris/x86, or Solaris/Sparc). If the program is running on a UNIX-like operating system the value true is returned, otherwise the value false is returned.

This function doesn't actually test the operating system the program is running on, instead it tests the edition of the interpreter that is running the program. It is assumed for example that the Linux version of the interpreter is running under Linux, and the Windows version of the interpreter is running under Windows.

Parameter

None.

Portability

Operating Systems: All
Standard Pascal: No

7.2.70 The upcase function

Description

The **upcase** function returns the uppercase letter corresponding to the parameter passed to it. **NOTE:** If there is no uppercase letter corresponding to the parameter passed to it then the parameter is returned unchanged. The type of the value returned by this function is always char.

Parameter

The **upcase** function's only parameter is an expression of char type.

Example

```
upcase('a')  returns  'A'
upcase('A')  returns  'A'
upcase('h')  returns  'H'
upcase('1')  returns  '1'
upcase('+')  returns  '+'
upcase('_')  returns  '_'
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.71 The uppercase function

Description

The **uppercase** function returns a string formed by converting all lowercase characters in the parameter passed to it, into uppercase characters, and leaving all non-lowercase characters in the parameter unchanged.

Parameter

The **uppercase** function's only parameter is an expression of string type or char type.

Example

```
uppercase('Hello!!')  returns   'HELLO!!'
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.72 The urldecode function

Description

The **urldecode** function returns a string formed by decoding the URL encoded string passed as it's parameter. URL decoding is a simple two step process. First all plus characters (**'+'**) are converted to spaces (**' '**). Second all hexadecimal sequences like **'%xx'** are converted to the equivalent binary character.

This function can be useful in Common Gateway Interface (CGI) programs for processing GET or POST methods, since these methods URL encode strings.

Parameter

The **urldecode** function's only parameter is an expression of string type or char type.

Example

```
urldecode('irie+pascal')            returns   'irie pascal'
urldecode('%2Fusr%2Flocal%2Fbin')   returns  '/usr/local/bin'
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.73 The version function

Description

The **version** function returns the version number of the interpreter running the program. The version number returned by this function is constructed of four bytes in little-endian format as follows:

1. The first byte contains the least significant verion number, and is changed when very minor changes are made to the interpreter (e.g. minor bug fixes).
2. The second byte contains the minor version number, and is changed when significant changes are made to the interpreter (e.g. support for new features are added).
3. The third byte contains the major version number, and is changed when major changes are made to interpreter (e.g. the format of executable is changed).
4. The fourth byte is currently undefined and always contains zero.

The type of the value returned by this function is always integer.

The version number returned by this function is **not** related to the version of the Irie Pascal compiler or the version of the operating system.

Currently you will probably not need to use this function, however this function is provided because each new version of the interpreter is likely to add support for new features. Therefore programs in the future may need to know which version of the interpreter they are running on to determine whether a particular feature is available (see the supported function for a better way to do this).

Parameter

None.

Example

```
version  currently returns the following bytes 00 05 02 00
```

So, the major version number is five, the minor version number is two, and the least significant version number is zero.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.2.74 The wait function

Description

The **wait** function accepts one or more parameters as described below, and waits (i.e. suspends execution of the program) until signalled to return, or until a timeout period expires.

Parameters

- The first parameter is an expression of integral type, which specifies the timeout period in milliseconds. If the value of the first parameter is zero then the **wait** function doesn't actually wait, instead it just checks whether it has been signaled to return. If the value of the first argument is equal to the built-in contant maxword then the timout interval is infinite, and the **wait** function will wait indefinitely until signalled to return.
- The other parameters, if supplied, are expressions of integral type, which specify handles to *synchronization objects*. The **wait** function will wait until one of the *synchronization objects* signals it to return, or until the timeout interval expires. The **wait** function returns the handle to the *synchronization object* that signaled it, or zero if the timeout period expires before it was signalled to return.

This function was implemented to make it easy to write well-behaved services (i.e. services that stop running when requested to do so). The stopserviceevent function returns a handle to a *synchronization object* that will signal when the program (if it is a service) is requested to stop running.

The authoritative source of information about *synchronization objects* is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

Example

For example, the following procedure was taken from the WinServS sample program and uses the **wait** function to stop running when requested to do so. **ProgramState** is a global variable used as a flag to signal when the program is stopping (i.e. being shut down) to prevent recursive calls to the **ServerShutDown** procedure. The **ServerShutDown** procedure shuts down the program.

```
procedure CheckForStopServiceEvent;
const
WAIT_INTERVAL=0;
begin
if (ProgramState<>Stopping) and (StopServiceEvent<>0) then
begin
//LogMessage('Calling wait');
if wait(WAIT_INTERVAL, StopServiceEvent)=StopServiceEvent then
begin
ProgramState := Stopping;
ServerShutDown;
end;
end
end;
```

Portability

Operating Systems: All
Standard Pascal: No

7.2.75 The wherex function

Description

The **wherex** function returns the text mode screen's current x coordinate (i.e. the column where the next character written to the screen will be displayed). **NOTE:** Screen columns are numbered from one (not zero).

Parameter

None.

Portability

Operating Systems: Windows, and DOS
Standard Pascal: No

7.2.76 The wherey function

Description

The **wherey** function returns the text mode screen's current y coordinate (i.e. the row where the next character written to the screen will be displayed). **NOTE:** Screen rows are numbered from one (not zero).

Parameter

None.

Portability

```
Operating Systems:Windows and DOS
Standard Pascal:No
```

7.3.1 Function declarations

Description

Function identifiers are identifiers that have been associated with a function using a function declaration.

Example

Here is an example of a function declaration.

```
//This function checks whether a files exists. Actually it really checks
//to see if a file can be opened for reading without causing an error,
//which is usually the same thing. The function first turns off run-time error
//trapping so that if an error occurs it can be retrieved with getlasterror
//rather than causing the program to halt. Then the function attempts to
//open the file for reading. If no errors occur then the file is closed and
//the file must exist. If an error occurs then the function assumes the file
//does not exist. In either case error trapping it turned back on before the
//function exits.
function FileExists(name : filename) : boolean;
var
f : text;
begin (* FileExists *)
traperrors(false);
reset(f, name);
if getlasterror=0 then
begin
close(f);
FileExists := true;
end
else
FileExists := false;
traperrors(true)
end; (* FileExists *)
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
function-declaration =
function-heading ';'directive |
function-identification ';'function-block |
function-heading ';'function-block
```

```
block =declarative-part statement-part

compound-statement ='begin'statement-sequence 'end'

declaration-group =
label-declaration-group |
constant-definition-group |
type-definition-group |
variable-declaration-group |
function-declaration |
procedure-declaration

declarative-part ={declaration-group }

directive =forward-directive |external-directive

formal-parameter-list ='('formal-parameter-section {';'formal-parameter-section }')'

formal-parameter-section =value-parameter-specification |
variable-parameter-specification |
procedure-parameter-specification |
function-parameter-specification

function-block =block

function-heading ='function'identifier [formal-parameter-list ]':'result-type

function-identification ='function'function-identifier

function-identifier =identifier

function-parameter-specification =function-heading

identifier =letter {letter |digit }

identifier-list =identifier {','identifier }

procedure-heading ='procedure'identifier [formal-parameter-list ]

procedure-identification ='procedure'procedure-identifier

procedure-identifier =identifier

procedure-parameter-specification =procedure-heading

result-type =type-identifier

statement-part =compound-statement

statement-sequence =statement {';'statement }

type-identifier =identifier

value-parameter-specification =identifier-list ':'type-identifier

variable-parameter-specification ='var'identifier-list ':'type-identifier
```

7.4.1 The built-in procedures

The **built-in procedures** are procedures that are automatically declared by the compiler, and each **built-in procedure** is associated with a built-in procedure identifier. Unlike reserved words, you can declare your own identifiers with the same spelling as a built-in procedure identifier, and if you do so then your declaration will override the declaration made by the compiler. The built-in procedure identifiers are listed below:

7.4.2 The append procedure

Description

The **append** procedure associates a file variable with a file (i.e. opens the file variable and allows it to be used to manipulate the file). The **append** procedure opens the file variable in append mode. The open procedure can also be used to open file variables in append mode.

Parameters

1. The first parameter is a reference to the file variable to be opened.
2. The second parameter is an expression of string type or char type, and *names* the file variable, referenced by the first parameter, as if the built-in procedure assign had been called.

The name of the file variable, referenced by the first parmater, controls which file is opened by the **append** procedure. If the name is a non-empty string then a file with that name is opened. If the name is an empty string then the file opened is either the standard output file, or a temporary file (with a system generated name), depending on the project options set when the program was compiled.

File variables can get *named* in the following ways:

- By calling the built-in procedure assign.
- By supplying the second parameter when calling the built-in procedures **append**, open, popen, reset, or rewrite.
- By using the file variables as program parameters.

Example

```
append(f, 'file.txt')   opens a file named 'file.txt'
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.3 The assert procedure

Description

The **assert** procedure is based on the C language macro of the same name. The **assert** procedure can be enabled/disabled using a compiler option.

When enabled the **assert** procedure evaluates its parameter, which must be a boolean expression. If the expression evaluates to false then the program is terminated. If the expression evaluates to true then the procedure does nothing.

When disabled the **assert** procedure does nothing.

The idea behind the **assert** procedure is that you use it during debugging to test the values in your program, and after your program is debugged you don't have to search your program and remove all the calls to this procedure, instead you just disable it using a compiler option.

You should use the **assert** procedure only to test conditions that **must** be true if your program has no bugs. A common mistake is using the **assert** procedure to test normal error conditions that may happen even if your program is bug-free (such as out of memory conditions or invalid user input). For normal error conditions it is usually better to check for these in some other way and handle them in a more graceful way.

Parameters

The **assert** procedure's only parameter is an expression of boolean type.

Portability

`Operating Systems:``All`
`Standard Pascal:``No`

7.4.4 The assign procedure

Description

The **assign** procedure *names* a file variable. **NOTE:** You can't use this function to rename a file, use the built-in procedure rename if you want to do that.

Parameters

1. The first parameter is a reference to the file variable to be *named*.
2. The second parameter is an expression of string type or char type, and is the new *name* of the file variable.

The name of a file variable controls which file is opened when the file variable is opened. If the name is a non-empty string then a file with that name is opened. If the name is an empty string then the file opened is either the standard output file, or a temporary file (with a system generated name), depending on the project options set when the program was compiled.

Example

For example to open a file named 'README.TXT' for reading you could use.

```
assign(f, 'README.TXT');
reset(f);
```

where **f** is a reference to a file variable.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.5 The chdir procedure

Description

The **chdir** procedure changes the current directory. **NOTE:** Directories are also called folders.

Parameter

The **chdir** procedure's only parameter is an expression of string type or char type which is the name of the directory to change to.

Example

```
chdir('\irie\samples') changes the current directory to '\irie\samples'
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.6 The close procedure

Description

The **close** procedure closes a file variable, and the file or pipe associated with it. Before the file variable is closed, it is flushed as if the built-in procedure flush was called. **NOTE:** The file variable must be open when this procedure is called.

Parameter

The **close** procedure's only parameter is a reference to the file variable to close.

Example

```
function CountLines(name : filename) : integer;
var
count : integer;
f : text;
begin
count := 0;
reset(f, name);
while not eof(f) do
begin
readln(f);
inc(count);
end;
CountLines := count;
close(f)
end;
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.7 The closedir procedure

Description

The **closedir** procedure closes a directory variable (i.e. destroys the association between a directory variable and it's directory). **NOTE:** Directories are also called folders.

Parameter

The **closedir** procedure's only parameter is a reference to the directory variable to close.

Example

```
//Below is a simple program that lists all the files in the
//current directory.
program listfiles(output);
var
d : dir;
filename : string;
begin
opendir(d, '.');  // Open current directory
repeat
readdir(d, filename);  // Get next file in directory
if filename <> '' then
writeln(filename);
until filename = '';
closedir(d) // Close directory
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.8 The clrscr procedure

Description

The **clrscr** procedure clears the screen and moves the cursor to the top left corner of the screen.

Parameter

None.

Portability

Operating Systems: All
Standard Pascal: No

7.4.9 The crc32 procedure

Description

The **crc32** procedure computes a 32 bit Cyclic Redundancy Check (CRC).

Parameter

1. The first parameter is a reference to a variable of integer type or word type which is used to accumulate the result.
2. The second parameter is an expression of byte type or char type, and is the value to use as the data.
3. The third parameter is an expression of integral type, and is the value to use as the CRC polynomial. If this parameter is not supplied then the default CRC polynomial (i.e. $04C11DB7) is used.

Example

```
//*****************************************************
// This program uses the built-in procedure crc32
//  to calculate the 32-bit CRC for a file.
// Basically the program asks for the name of the file
// and then it reads the file a character at a time
// passing each byte to crc32.
// Finally it calls crc32 4 times with zero to
//  complete the crc calculation. This is part of
//  the process of calculating CRCs.
//*****************************************************
```

```
program filecrc(input, output);
var
fn : filename;
f : file of char;
c : char;
crc : integer;
begin
write('Enter filename: ');
readln(fn);
writeln('Calculating CRC...');
crc := 0;
reset(f, fn);
while not eof(f) do
begin
read(f, c);
crc32(crc, c);
end;

crc32(crc, 0);
crc32(crc, 0);
crc32(crc, 0);
crc32(crc, 0);

writeln('CRC for ', fn, ' is ', crc, ' (', hex(crc), ')');
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.10 The dec procedure

Description

The **dec** procedure decrements the value stored in a variable.

Parameter

1. The first parameter is a reference to the variable of ordinal type whose value is to be decremented.
2. The second parameter is an expression of integral type, and is the value by which the variable is to be decremented. If this parameter is not supplied then the variable is decremented by one.

Example

```
dec(x)      decrements the variable "x" by 1
dec(x, 10)  decrements the variable "x" by 10
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.11 The delay procedure

Description

The **delay** procedure waits for a specified number of milli-seconds to elapse.

Parameter

The **delay** procedure's only parameter is an expression of integral type and is the number of milli-seconds that the program should wait.

Example

```
delay(1000)  makes the program wait for 1000 milli-seconds (i.e. 1 second).
```

Portability

Operating Systems: Windows
Standard Pascal: No

7.4.12 The delete procedure

Description

The **delete** procedure deletes components from a list or a string.

Parameter

1. The first parameter is a reference to the list variable, or the string variable from which components are deleted.
2. The second parameter is an expression of integral type, and specifies the position to start deleting components from.
3. The third parameter is an expression of integral type, and specifies the number of components to delete. If this parameter is omitted then all the components from the start position to the end of the list or string are deleted.

Example

```
delete(l, 1)    deletes all components in the list l
delete(l, 2)    deletes all components from lexcept the first one
delete(l, 1, 3) deletes the first three components from l
delete(s, 1)    deletes all characters from the string s
delete(s, 2)    deletes all characters from sexcept the first one
delete(s, 1, 3) deletes the first three characters from s
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.13 The dispose procedure

Description

The **dispose** procedure destroys an *identified variable* (associated with a pointer variable), or all of the *selected variables* (associated with a list variable), or an *instance variable* (associated with an object variable).

The **dispose** procedure will store an undefined value into the pointer variable, or the list variable, or the object variable.

Parameter

The **dispose** procedure's only parameter is a reference to a pointer variable, or a list variable, or a object variable. It is an error if the variable referenced by the parameter contains the special value nil or is undefined.

Portability

Operating Systems: All
Standard Pascal: Yes

In Standard Pascal, the **dispose** procedure's parameter is an expression of pointer type, and not a reference to a pointer variable as required by Irie Pascal.

Standard Pascal does not support list types or object types and so does not support using the **dispose** procedure on variables of those types.

7.4.14 The erase procedure

Description

The **erase** procedure deletes a file.

Parameter

The **erase** procedure's only parameter is a reference to a file variable. This file variable must be *named* (see the built-in procedure assign), and can not be open.

Example

```
//*************************************************
// This program removes (i.e. erases) a file
// First it asks for the name of the file
// Then it assigns the name to a file variable
// Finally it calls 'erase' on the file variable
//************************************************
program rm(input, output);
var
fn : filename;
f : binary;
begin
write('Which file do you want to delete? ');
readln(fn);
assign(f, fn);
erase(f);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.15 The exec procedure

Description

The **exec** procedure executes a command by passing it to the command processor. The built-in variable exitcode contains the numeric code returned by the command processor to indicate whether an error occured. If return value of zero means that no error occured while executing the command, and any other return value identifies the particular error that occured. The values used by the command processors of different operating system, to identify which error occured, are not standardized and is not documented here.

Parameter

1. The first parameter is an expression of string type or char type, and is the command to execute.
2. The second parameter is an expression of string type or char type, and contains the arguments used by the command. If no arguments are used by the command then this parameter is an empty string.

Example

```
//***************************************************
// This program lists all the files ending with .pas
//  in the current diectory using the built-in
//  procedure "exec".
// Basically the program determines whether it is running
//  under a Unix-like platform or not and passes
//  the appropráte list command to "exec" along
//  with the "*.pas" argument so that only files
//  ending with .pas are listed.
program dirpas(output);
```

```
const
UnixListCommand = 'ls -l';
OtherListCommand = 'dir';

begin
if UnixPlatform then
exec(UnixListCommand, '*.pas')
else
exec(OtherListCommand, '*.pas');
if exitcode <> 0 then
writeln('Error listing directory', exitcode)
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.16 The exit procedure

Description

The **exit** procedure terminates the program or the function or procedure in which it is used, and optionally may return a value to the caller.

If used in a function then the **exit** procedure terminates the function. If used in a function named **f** then **exit(e)** is equivalent to:

```
f := e;
exit
```

If used in a procedure then **exit** terminates the procedure. If used in a procedure then **exit(x)** is invalid since you can not return a value from a procedure.

If used in the main program then **exit** terminates the program. If used in the main program then **exit(e)** is equivalent to:

```
ExitCode := e;
exit
```

(i.e. the program is terminated with **e** as the exit code).

Parameter

The **exit** procedure's only paramter is an expression. If supplied this parameter is the value to be returned by the program or function or procedure.

Portability

Operating Systems: All
Standard Pascal: No

7.4.17 The fill procedure

Description

The **fill** procedure fills all the bytes in a variable with a specified value.

Parameter

1. The first parameter is a reference to the variable, whose bytes are to be filled.
2. The second parameter is an expression of integral type, and specifies the value used to fill the bytes in the variable.

Portability

Operating Systems: All
Standard Pascal: No

7.4.18 The flush procedure

Description

The **flush** procedure flushes a file or pipe (i.e. all data waiting in memory to be written is written).

Parameter

The **flush** procedure's only parameter is a reference to the file variable associated with the file or pipe to be flushed. The file variable must be open before calling this function.

Portability

Operating Systems: All
Standard Pascal: No

7.4.19 The fsplit procedure

Description

The **fsplit** procedure splits a filename into its component parts.

Parameter

1. The first parameter is an expression of string type or char type which contains the filename to be split.

2. The second parameter is optional, and if supplied is a reference to the string variable which will store the directory/folder part of the filename. If this parameter is not supplied then the directory/folder part of the filename is not stored.
3. The third parameter is optional, and if supplied is a reference to the string variable which will store the name part of the filename. If this parameter is not supplied then the name part of the filename is not stored.
4. The fourth parameter is optional, and if supplied is a reference to the string variable which will store the extension part of the filename. If this parameter is not supplied then the extension part of the filename is not stored.

Although the second, third, and fourth parameters are optional, at least one of them must be supplied.

Example

```
fsplit('/usr/local/bin/readme.txt', d, n, e) stores
'/usr/local/bin' in "d" and
'readme' in "n" and
'.txt' in "e"

fsplit('c:\autoexec.bat', d,,) stores 'c:\' in "d"
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.20 The get procedure

Description

The **get** procedure reads the next component from a file into the file's buffer variable. **NOTE:** This procedure is seldom used since it is usually easier to use the read or readln procedures.

Parameter

The **get** procedure's only parameter is a reference to the file variable associated with the file to be read. The file variable must be open before calling this function.

Example

```
//*********************************************************
//This program is similar to the unix shell command "cat".
//It uses the built-in procedures "get" and "put" as well
// as the file buffer's associated with the input and
// output file to copy the contents of the input file to
// the output file.
program cat(f, g);
var
f, g : text;
c : char;
begin
reset(f);
```

```
rewrite(g);
while not eof(f) do
begin
c := f^;
get(f);
g^ := c;
put(g);
end;
close(g);
close(f);
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.4.21 The getdate procedure

Description

The **getdate** procedure retrieves the current system date.

Parameter

1. The first parameter is a reference to a variable of integer type or word type which will store the year part of the current system date.
2. The second parameter is a reference to a variable of integer type or word type which will store the month part (1 to 12) of the current system date.
3. The third parameter is a reference to a variable of integer type or word type which will store the day part (1 to 21) of the current system date.
4. The fourth parameter is a reference to a variable of integer type or word type which will store the day of the week part (0 to 6 counting from Sunday) of the current system date.

Example

Suppose the current system date is Wednesday September 30, 1998 then after:

```
GetDate(year, month, day, day_of_week)

year = 1998
month = 9
day = 30
day_of_week = 3
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.22 The getfiledate procedure

Description

The **getfiledate** procedure retrieves the modification date of a file.

Parameter

1. The first parameter is an expression of string type or char type which is the name of the file whose modification date is to be retrieved.
2. The second parameter is a reference to a variable of integer type or word type which will store the year part of the date the file was last modified.
3. The third parameter is a reference to a variable of integer type or word type which will store the month part (1 to 12) of the date the file was last modified.
4. The fourth parameter is a reference to a variable of integer type or word type which will store the day part (1 to 31) of the date the file was last modified.

Example

Suppose the file 'c:\irie\readme.txt' was last modified on Wednesday September 30, 1998 then after:

```
getfiledate('c:\irie\readme.txt', year, month, day)

year = 1998
month = 9
day = 30
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.23 The getfilemode procedure

Description

The **getfilemode** procedure retrieves the permission mode of a file. The permission mode can be AND'd with the permission constants to determine the permissions of the file.

Parameter

1. The first parameter is an expression of string type or char type which is the name of the file whose permission mode is to be retrieved.
2. The second parameter is a reference to a variable of integer type or word type which will store the permission mode of the file.

Example

```
//*************************************************************
//This program uses the built-in procedure "getfilemode" to
// determine if a file is a directory.
program isdirectory(input, output);
var
s : filename;
mode : integer;
isdir : boolean;
begin
write('Enter filename:');
readln(s);
s := fexpand(s);
getfilemode(s, mode);
isdir := (mode and dir_bit) <> 0;
if isdir then
writeln(s, ' is a directory')
else
writeln(s, ' is not a directory');
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.24 The getfiletime procedure

Description

The **getfiletime** procedure retrieves the modification time of a file.

Parameter

1. The first parameter is an expression of string type or char type which is the name of the file whose modification time is to be retrieved.
2. The second parameter is a reference to a variable of integer type or word type which will store the hour the file was last modified.
3. The third parameter is a reference to a variable of integer type or word type which will store the minute the file was last modified.
4. The fourth parameter is a reference to a variable of integer type or word type which will store the second the file was last modified.

Example

Suppose the file 'c:\irie\readme.txt' was last modified at 2:45:05 P.M. then after:

```
getfiletime('c:\irie\readme.txt', hour, minute, second)

hour = 14
minute = 45
second = 5
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.25 The gettime procedure

Description

The **gettime** procedure retrieves the current system time.

Parameter

1. The first parameter is a reference to a variable of integer type or word type which will store the current hour.
2. The second parameter is a reference to a variable of integer type or word type which will store the current minute.
3. The third parameter is a reference to a variable of integer type or word type which will store the current second.

Example

Suppose the current system time is 2:45:05 P.M. then after:

```
gettime(hour, minute, second)

hour = 14
minute = 45
second = 5
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.26 The gotoxy procedure

Description

The **gotoxy** procedure moves the text mode cursor to a specified (x, y) coordinate on the screen.

Parameter

1. The first parameter is an expression of integral type which specifies the x coordinate to move the cursor to.
2. The second parameter is an expression of integral type which specifies the y coordinate to move the cursor to.

Example

```
gotoxy(1,1)  moves the cursor to the top left-hand corner of the screen.
```

Portability

Operating Systems: Windows and DOS
Standard Pascal: No

7.4.27 The halt procedure

Description

The **halt** procedure terminates the program and optionally returns an integral value to the program's caller.

Parameter

The **halt** procedure's only paramter is optional, and if supplied is an expression of integral type that is the value returned by the program.

Portability

Operating Systems: All
Standard Pascal: No

7.4.28 The inc procedure

Description

The **inc** procedure increments the value stored in a variable.

Parameter

1. The first parameter is a reference to the variable of ordinal type whose value is to be incremented.
2. The second parameter is an expression of integral type, and is the value by which the variable is to be incremented. If this parameter is not supplied then the variable is incremented by one.

Example

```
inc(x)      decrements the variable "x" by 1
inc(x, 10)  decrements the variable "x" by 10
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.29 The insert procedure

Description

The **insert** procedure either inserts a string into another string or inserts a value into a list.

Parameter

1. The first parameter is an expression whose value is to be inserted into the string or list.
2. The second parameter is a reference to the string variable or the list variable into which the value is to be inserted.
3. The third parameter is an expression of integral type, and is the position in the string or list where the value is to be inserted. If this parameter is not supplied then the value is inserted at the end of the string or list.

Example

Here are some examples using the **insert** procedure with strings:

Assuming that **x** is a string variable then after:

```
x := "ABC";
insert('DEF', x); //insert 'DEF' at the end of "x"
```

x will contain **'ABCDEF'**, and after:

```
x := 'ABFGHI';
insert('CDE', x, 3); //insert 'CDE' at the third position
```

x will contain **'ABCDEFGHI'**.

Portability

Operating Systems: All
Standard Pascal: No

7.4.30 The intr procedure

Description

The **intr** procedure generates a x86 processor interrupt.

Parameter

1. The first parameter is an expression of integral type whose value is the number of the interupt to generate (0-255).
2. The second parameter is a reference to the variable of registers type which contains values to pass into the x86 processor registers before the interrupt is generated, and which receives the contents of the x86 processor registers after the interrupt is serviced.

Example

```
//For example the following simple program uses interrupt $10
//to clear the screen.
program cls;
var
regs : registers;
begin
fill(regs, 0);
regs.eax := $0002;
intr($10, regs)
end.
```

Portability

```
Operating Systems:DOS only
Standard Pascal:No
```

7.4.31 The mkdir procedure

Description

The **mkdir** procedure creates a new directory/folder. If the directory/folder already exists then the **mkdir** procedure call will fail.

Parameter

1. The first parameter is an expression of string type or char type which is the name of the directory/folder to create. If the directory/folder being created is a sub-directory/sub-folder then the parent directories/folders must exist.
2. The second parameter is optional, and if supplied is an expression of integral type which specifies permissions for the directory/folder being created. The permission constants can be used to specify directory/folder permissions. If this parameter is not supplied then the default permissions are used. **NOTE:** Some operating systems like Windows ignore this parameter.

Example

The following call to create the directory/folder **c:\irie\test** will fail if the directory/folder **c:\irie** does not already exist.

```
mkdir('\irie\test') makes the directory 'test' in
the directory '\irie'
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.32 The move procedure

Description

The **move** procedure copies memory from one memory address to another. The memory is copied correctly even if the source and destination memory locations overlap.

Parameter

1. The first parameter is a reference to a variable that specifies the source memory address (i.e. the address of the first byte to be copied). If the variable is of address type or of pointer type then the address stored in the variable is the source memory address. If the variable is **not** of address type or of pointer type then the address of the variable itself is the source memory address.
2. The second parameter is a reference to a variable that specifies the destination memory address (i.e. the address where the memory is to be copied). If the variable is of address type or of pointer type then the address stored in the variable is the destination memory address. If the variable is **not** of address type or of pointer type then the address of the variable itself is the destination memory address.
3. The third parameter is an expression of integral type which specifies the number of memory locations to copy. **NOTE:** This parameter is always treated as an unsigned integer number.

Portability

Operating Systems: All
Standard Pascal: No

7.4.33 The msdos procedure

Description

The **msdos** procedure makes a call to MSDOS through x86 processor interrupt $21.

Parameter

The **msdos** procedure's only parameter is a reference to the variable of registers type which contains values to pass into the x86 processor registers before the interrupt is generated, and which receives the contents of the x86 processor registers after the interrupt is serviced.

Example

```
//For example the following simple program uses
```

```
// MSDOS function $30 to get the MSDOS version number.
program GetVer(output);
var
regs : registers;
begin
fill(regs, 0);
regs.ah := $30;
regs.al := $00;
msdos(regs);
write('MSDOS version is ');
if regs.al = 0 then
writeln('1')
else
writeln(regs.al:1, '.', regs.ah:1)
end.
```

Portability

Operating Systems:DOS Only
Standard Pascal: No

7.4.34 The new procedure

Description

The **new** procedure performs the following actions:

- Creates an *identified variable* from a pointer variable and stores the value identifying the *identified variable* (i.e. the address of the *identified variable*) in the pointer variable.
- Initializes a list variable before it is used.
- Creates an *instance variable* from an object variable and stores the value referencing the *instance variable* (i.e. the address of the *instance variable*) in the object variable.

Parameter

The **new** procedure's only parameter is a reference to a pointer variable, object variable, or object variable.

Portability

Operating Systems: All
Standard Pascal: Yes

Standard Pascal does not support the list type or the object type and so does not support using the **new** prococedure on variables of those types.

7.4.35 The open procedure

Description

The **open** procedure associates a file variable with a file (i.e. opens the file variable and allows it to be used to manipulate the file). The **open** procedure can open the file variable in read-only mode, write-only mode, read/write mode, or append mode.

Parameters

1. The first parameter is a reference to the file variable to be opened.
2. The second parameter is an expression of string type or char type, and *names* the file variable, referenced by the first parameter, as if the built-in procedure assign had been called.
3. The third parameter is an expression of integral type which specifies the mode the file variable should be opened in. The file mode constants can be used to specify the mode. The readmode and writemode constants can be added or OR'd together to specify that the file variable be opened in read/write mode (i.e. for both reading and writing).

The name of the file variable, referenced by the first parmater, controls which file is opened by the **open** procedure. If the name is a non-empty string then a file with that name is opened. If the name is an empty string then the file opened is either the standard output file, or a temporary file (with a system generated name), depending on the project options set when the program was compiled.

File variables can get *named* in the following ways:

- By calling the built-in procedure assign.
- By supplying the second parameter when calling the built-in procedures **open**, append, popen, reset, or rewrite.
- By using the file variables as program parameters.

Example

```
open(f, 'filename', readmode);   //this is the same as reset(f, 'filename');
open(f, 'filename', writemode); //this is the same as rewrite(f, 'filename');
open(f, 'filename', appendmode); //this is the same as append(f, 'filename');
```

The following call to **open**

```
open(f, 'test.txt', readmode+writemode);
```

will open the file **'test.txt** in read/write mode (i.e. for both reading and writing).

Portability

Operating Systems: All
Standard Pascal: No

7.4.36 The opendir procedure

Description

The **opendir** procedure opens a directory variable (i.e. associates the directory variable with a directory).
NOTE: Directories are also called folders.

Parameters

- The first parameter is a reference to the directory variable to open.
- The second parameter is an expression of string type or char type, and is the name of the directory to associate with the directory variable.

Example

```
//Below is a simple program that lists all the files in the
//current directory.
program listfiles(output);
var
d : dir;
filename : string;
begin
opendir(d, '.');  // Open current directory
repeat
readdir(d, filename);  // Get next file in directory
if filename <> '' then
writeln(filename);
until filename = '';
closedir(d) // Close directory
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.37 The pack procedure

Description

The **pack** procedure assigns values to all the elements of a packed array by copying some or all of the elements of an unpacked array into the packed array. The elements of both arrays must have the same type.

Parameter

- The first parameter is a reference to the unpacked array variable that contains the array elements to copy.
- The second parameter is an expression of ordinal type that specifies the start element of the unpacked array (i.e. the element of the unpacked array to start copying from). The elements from

the start element of the unpacked array to the last element of the unpacked array are available to be copied into the packed array. It is an error if the number of elements in the unpacked array available to be copied into the packed array is less then the number of elements in the packed array, since values must be copied to all the elements in the packed array. **NOTE:** This parameter must be assignment compatible with the index type of the unpacked array.
- The third parameter is a reference to the packed array variable that contains the array elements to be copied into.

Example

Suppose you have the following arrays

```
p : packed array[21..23] of integer;
u : array[1..10] of integer;
```

then

```
pack(u, 1, p)
```

copies elements 1 to 3 of **u** into elements 21 to 23 of **p**.

And

```
pack(u, 5, p)
```

copies elements 5 to 7 of **u** into elements 21 to 23 of **p**

```
pack(u, 9, p)
```

is an error because only two elements of **u** (i.e. 9 and 10) are available to be copied but **p** has three elements.

Portability

Operating Systems: All
Standard Pascal: Yes

7.4.38 The page procedure

Description

The **page** procedure writes a form-feed control character to a file variable of type text. According to Standard Pascal, this procedure is supposed to affect the text file in such a way that subsequent output appears on a new page.

Parameter

The **page** procedure's only parameter is a reference to the file variable of type text to which the form-feed control character will be written.

Portability

Operating Systems: All
Standard Pascal: Yes

7.4.39 The popen procedure

Description

The **popen** procedure creates a process, opens a pipe (i.e. a communications channel) to the process, and associates a file variable with the pipe (i.e. opens the file variable). The **popen** procedure can open the file variable in read-only mode, write-only mode, or read/write mode.

Read-Only Mode

If the file variable is opened in read-only mode then the program can read the pipe and retrieve values written to the pipe by the process. However the program can not write values through the pipe for the process to read.

Write-Only Mode

If the file variable is opened in write-only mode then the program can write values to the pipe for the process to read. However the program can not read the pipe to retrieve values written to the pipe by the process.

Read/Write Mode

If the file variable is opened in read/write mode then the program can write values to the pipe for the process to read, and read the pipe to retrieve values written to the pipe by the process.

Parameter

1. The first parameter is a reference to the file variable to be opened.
2. The second parameter is an expression of string type or char type, and is the program from which the process is created.
3. The third parameter is an expression of integral type which specifies the mode the file variable should be opened in. The file mode constants can be used to specify the mode. The readmode and writemode constants can be added or OR'd together to specify that the file variable be opened in read/write mode (i.e. for both reading and writing).

Portability

Operating Systems: All
Standard Pascal: No

7.4.40 The put procedure

Description

The **put** procedure writes the contents of a file's buffer variable to the file and empties the file's buffer variable leaving it undefined. **NOTE:** This procedure is seldom used since it is usually easier to use the write or writeln procedures.

Parameter

The **put** procedure's only parameter is a reference to the file variable associated with the file to be written to. The file variable must be open before calling this function.

Example

The simple program below uses the built-in procedure **put** to write a message to the standard output.

```
program hello2(output);
begin
output^ := 'H';
put(output);
output^ := 'e';
put(output);
output^ := 'l';
put(output);
output^ := 'l';
put(output);
output^ := 'o';
put(output);
output^ := ' ';
put(output);
output^ := 'w';
put(output);
output^ := 'o';
put(output);
output^ := 'r';
put(output);
output^ := 'l';
put(output);
output^ := 'd';
put(output);
output^ := '!';
put(output);
output^ := '!';
put(output);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.41 The randomize procedure

Description

The **randomize** procedure initializes the random number generator with a seed. The random number generator is a psuedo random number generator (i.e. the sequence of numbers generated appears to be random but the same sequence of numbers is generated every time the same seed is used).

Parameter

The **randomize** procedure's only parameter is optional, and if supplied is an expression of integral type that is the seed used to initialize the random number generator. If this parameter is not supplied then the random number generator is initialized with a seed derived from the current system date and time. Specifying this parameter is useful if you want a repeatable sequence of random numbers. Or you can just not call the **randomize** at all in which case the the random number generator generates the same sequence of the random numbers, every time your program is run.

Example

The following program generates two random sequences of numbers. The first sequence repeats every time the program is run since the random number generated is initialed with the same seed each time the program is run. The second sequence does not repeat since the random number generator is initialized with a system generated seed each time the program is run.

```
program rnd(output);
var
i : integer;
begin
writeln('Repeatable sequence');
randomize(6566);
for i := 1 to 10 do
writeln(random(10));
writeln('Non-repeatable sequence');
randomize;
for i := 1 to 10 do
writeln(random(10));
end.
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.42 The rawread procedure

Description

The **rawread** procedure reads up to a specified number of characters from a file into a character buffer. The actual number of characters read by this procedure may be less than the maximum number of characters specified if there is an I/O error (you can check this by disabling I/O checking and using the getlasterror function) or if the end-of-file is reached.

Parameter

1. The first parameter is a reference to the file variable associated with the file to read.
2. The second parameter is a reference to the character buffer used to store the characers read from the file. The character buffer must be a variable of type string or a packed array of type char.
3. The third parameter is an expression of integral type which specifies the maximum number of characters to read from the file.
4. The fourth parameter is a reference to a variable of type integer or type word which will contain the actual number of characters read from the file.

Example

The simple program below copies the contents of an input file into an output file. It is similar to the UNIX program "cat". The program uses the built-in procedures "rawread", and "rawwrite" to perform the I/O.

```
program cat(f, g);
const
max = 400;
var
f, g : text;
buffer : packed array[1..max] of char;
actual : integer;
begin
reset(f);
rewrite(g);
while not eof(f) do
begin
rawread(f, buffer, max, actual);
if actual <> 0 then
rawwrite(g, buffer, actual, actual);
end;
close(g);
close(f);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.43 The rawwrite procedure

Description

The **rawwrite** procedure writes up to a specified number of characters from a character buffer to a file. The actual number of characters written by this procedure may be less than the maximum number of characters specified if there is an I/O error (you can check this by disabling I/O checking and using the getlasterror function).

Parameter

1. The first parameter is a reference to the file variable associated with the file to write.
2. The second parameter is a reference to the character buffer used to store the characers to be written to the file. The character buffer must be a variable of type string or a packed array of type char.
3. The third parameter is an expression of integral type which specifies the maximum number of characters to be written to the file.
4. The fourth parameter is a reference to a variable of type integer or type word which will contain the actual number of characters written to the file.

Example

The simple program below copies the contents of an input file into an output file. It is similar to the UNIX program "cat". The program uses the built-in procedures "rawread", and "rawwrite" to perform the I/O.

```
program cat(f, g);
const
max = 400;
var
f, g : text;
buffer : packed array[1..max] of char;
actual : integer;
begin
reset(f);
rewrite(g);
while not eof(f) do
begin
rawread(f, buffer, max, actual);
if actual <> 0 then
rawwrite(g, buffer, actual, actual);
end;
close(g);
close(f);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.44 The read procedure

Description

The **read** procedure reads values from the file associated with a file variable into one or more variables.

Parameters

1. The first parameter is optional, and if supplied is a reference to the file variable associated with the file to read from. If this parameter is not supplied then the **read** procedure reads values from the file associated with input.

2. The other parameters are references to the variables used to store the values read from the file.

Reading From Text Files

If the **read** procedure reads values from a text file (i.e. a file associated with a file variable of type text), then the way the values are read is determined by the type of the variable used to store the values.

- If a variable of char type or a subrange of char type is being used to store the values read from the text file then a single character is read from the text file and this character is stored without conversion in the variable.
- If a variable of string type is being used to store the values read from the text file then characters are read from the text file and stored in the variable until either the end of the line is reached or the number of characters stored in the variable is equal to the maximum number of characters that can be stored in the variable.
- If a variable of an integral type or a subrange of an integral type is being used to store the values read from the text file then a sequence of characters forming a signed decimal integer number, possibly with leading blanks, is read from the file, and converted into the equivalent value of integer type before being stored in the variable.
- If a variable of real type, single type, or double type is being used to store the values read from the text file then a sequence of characters forming a signed real number, possibly with leading blanks, is read from the file, and converted into the equivalent value of real type before being stored in the variable.

Reading From Non-Text Files

If the **read** procedure reads values from a non-text file (i.e. a file associated with a file variable that is **not** of type text) then the type of the values read from the file is the same as the type of the variables used to store the values, and this type is usually the same as the file type's component type. The values are read from the file and stored in the variables without conversion.

Example

```
read(f, x, y, z)     reads three values from the file associated
with "f" into "x", "y", and "z".
```

Portability

Operating Systems: All
Standard Pascal: Yes

Standard Pascal does not support reading from text files into variable of string type.

7.4.45 The readdir procedure

Description

The **readdir** procedure reads a filename from a directory variable (i.e. retrieves the name of the next file in the directory associated with the directory variable). **NOTE:** Directories are also called folders.

Parameters

- The first parameter is a reference to the directory variable to read from.
- The second parameter is a reference to the string variable that will store the filename read from the directory variable.

Example

The simple program below opens the current directory and then uses "readdir" to read the names of all the files in the directory, and then finally closes the directory when done.

```
program listdir(output);
var
d : dir;
fn : filename;

begin
opendir(d, '.');   Open current directory
repeat
readdir(d, fn);   Get next file in directory
if fn <> '' then
writeln(fn);
until fn = '';
closedir(d)  Close directory
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.46 The readln procedure

Description

The **readln** procedure reads values from the file associated with a file variable of type text into one or more variables, and then skips to the beginning of the next line.

Parameters

1. The first parameter is optional, and if supplied is a reference to the file variable associated with the file to read from. If this parameter is not supplied then the **readln** procedure reads values from the file associated with input.
2. The other parameters are optional, and if supplied are references to the variables used to store the values read from the file.

Reading From Text Files

The **readln** procedure always reads values from a text file (i.e. a file associated with a file variable of type text), and the way the values are read is determined by the type of the variable used to store the values.

- If a variable of char type or a subrange of char type is being used to store the values read from the text file then a single character is read from the text file and this character is stored without conversion in the variable.
- If a variable of string type is being used to store the values read from the text file then characters are read from the text file and stored in the variable until either the end of the line is reached or the number of characters stored in the variable is equal to the maximum number of characters that can be stored in the variable.
- If a variable of an integral type or a subrange of an integral type is being used to store the values read from the text file then a sequence of characters forming a signed decimal integer number, possibly with leading blanks, is read from the file, and converted into the equivalent value of integer type before being stored in the variable.
- If a variable of real type, single type, or double type is being used to store the values read from the text file then a sequence of characters forming a signed real number, possibly with leading blanks, is read from the file, and converted into the equivalent value of real type before being stored in the variable.

Example

For example the program "batch.pas" below is a very primitive batch processor. It reads lines from a text file using "readln" and send the lines to the command processor to be executed.

```
program batch(f, output);
var
f : text;
s : string;
err : integer;
begin
reset(f);     (* open input file or standard input file *)
while not eof(f) do
begin
readln(f, s);
writeln('Executing ', s);
err := system(s);       (* Pass 's' to the command processor *)
writeln('Error code is ', err)
end
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

Standard Pascal does not support reading from text files into variable of string type.

7.4.47 The rename procedure

Description

The **rename** procedure changes the name of a file.

Parameters

1. The first parameter is a reference to the file variable that identifies the file to be renamed. The

name of the file variable is the name of the file to be renamed.
2. The second parameter is an expression of string type or char type, and is the new name of the file.

Example

```
(* This program renames a single file *)
program rename;
var
f : text;

procedure syntax;
begin
writeln('Renames a file');
writeln('Syntax: ivm rename old new');
writeln('  renames a file named ''old'' to ''new''');
exitcode := 1;
halt
end;

begin
if paramcount <> 2 then
syntax;
assign(f, paramstr(1)); //Specify which file you want to rename
rename(f, paramstr(2))  //Change the name of the file
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.48 The reset procedure

Description

The **reset** procedure associates a file variable with a file (i.e. opens the file variable and allows it to be used to manipulate the file). The **reset** procedure opens the file variable in read-only mode. The open procedure can also be used to open file variables in read-only mode.

Parameters

1. The first parameter is a reference to the file variable to be opened.
2. The second parameter is an expression of string type or char type, and *names* the file variable, referenced by the first parameter, as if the built-in procedure assign had been called. **NOTE:** This parameter is an extension to Standard Pascal.

The name of the file variable, referenced by the first parmater, controls which file is opened by the **reset** procedure. If the name is a non-empty string then a file with that name is opened. If the name is an empty string then the file opened is either the standard output file, or a temporary file (with a system generated name), depending on the project options set when the program was compiled.

File variables can get *named* in the following ways:

- By calling the built-in procedure assign.
- By supplying the second parameter when calling the built-in procedures append, open, popen, **reset**, or rewrite.
- By using the file variables as program parameters.

Example

The program below is used to count the number of vowels and the total number of characters in the input file. If you run this program and don't supply a program argument like

```
ivm vowels
```

then the program reads from the standad input file. If you run this program with a program argument like

```
ivm vowels filename
```

then the program will open the file named **filename** and read from it.

```
program vowels(f, output);
var
f : file of char;
tot, vow : integer;
c : char;
begin
reset(f);  //Open the input file in readmode.
tot := 0;  //Initialize total number of chars read so far
vow := 0;  //Initialize total number of vowels read so far
while not eof(f) do
begin
read(f, c);  //Read character
case c of
'a', 'e', 'i', 'o', 'u',
'A', 'E', 'I', 'O', 'U'
: vow := vow + 1;
otherwise //include "otherwise" so that non-vowels don't cause errors.
end;
tot := tot + 1;
end;
writeln('Total characters read = ', tot);
writeln('Vowels read = ', vow)
end.
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.4.49 The rewinddir procedure

Description

The **rewinddir** procedure rewinds a directory variable (i.e. alters the directory variable in such a way that filenames read from the directory variable start with the name of the first file in the directory associated with the directory variable). **NOTE:** Directories are also called folders.

Parameter

The **rewind** procedure's only parameter is a reference to a directory variable to rewind.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.50 The rewrite procedure

Description

The **rewrite** procedure associates a file variable with a file (i.e. opens the file variable and allows it to be used to manipulate the file). The **rewrite** procedure opens the file variable in write-only mode. The open procedure can also be used to open file variables in write-only mode.

Parameters

1. The first parameter is a reference to the file variable to open.
2. The second parameter is an expression of string type or char type, and *names* the file variable, referenced by the first parameter, as if the built-in procedure assign had been called. **NOTE:** This parameter is an extension to Standard Pascal.

The name of the file variable, referenced by the first parmater, controls which file is opened by the **rewrite** procedure. If the name is a non-empty string then a file with that name is opened. If the name is an empty string then the file opened is either the standard output file, or a temporary file (with a system generated name), depending on the project options set when the program was compiled.

File variables can get *named* in the following ways:

- By calling the built-in procedure assign.
- By supplying the second parameter when calling the built-in procedures append, open, popen, reset, or **rewrite**.
- By using the file variables as program parameters.

Portability

```
Operating Systems:All
Standard Pascal:Yes
```

7.4.51 The rmdir procedure

Description

The **rmdir** procedure removes (deletes) a directory/folder. If the directory/folder contains files then the **rmdir** call will fail.

Parameter

The **rmdir** procedure's only parameter is an expression of string type or char type that is the name of the directory/folder to remove.

Example

```
//************************************************
// This program removes (i.e. erases) a directory
// First it asks for the name of the directory
// Then it calls rmdir to remove the directory
//***********************************************
program rd(input, output);
var
fn : filename;
begin
write('Which directory do you want to delete? ');
readln(fn);
rmdir(fn);
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.52 The seek procedure

Description

The **seek** procedure moves the current file position of a file.

Parameters

1. The first parameter is a reference to the file variable associated with the file whose current file position is to be moved. **NOTE:** The file variable must be open in read-only mode or in read/write mode.
2. The second parameter is an expression of integral type and is the new file position of the file.

Portability

Operating Systems: All
Standard Pascal: No

7.4.53 The setfiledate procedure

Description

The **setfiledate** procedure sets the modification date of a file.

Parameters

1. The first parameter is an expression of string type or char type which is the name of the file whose modification date is to be set.
2. The second parameter is an expression of integral type that specifies the year part of the file modification date.
3. The third parameter is an expression of integral type that specifies the month part (1 to 12) of the file modification date.
4. The fourth parameter is an expression of integral typethat specifies the day part (1 to 31) of the file modification date.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.54 The setfiletime procedure

Description

The **setfiletime** procedure sets the modification time of a file.

Parameter

1. The first parameter is an expression of string type or char type which is the name of the file whose modification time is to be set.
2. The second parameter is an expression of integral type that specifies the hour part of the file modification time.
3. The third parameter is an expression of integral type that specifies the minute part of the file modification time.
4. The fourth parameter is an expression of integral typethat specifies the second part of the file modification time.

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.55 The sleep procedure

Description

The **sleep** procedure waits for a specified number of seconds to elapse.

Parameter

The **sleep** procedure's only parameter is an expression of integral type that is the number of seconds to wait.

Example

```
sleep(1);    (* waits for one (1) second. *)
```

Portability

```
Operating Systems:All
Standard Pascal:No
```

7.4.56 The str procedure

Description

The **str** procedure converts an integral or real value into a string and stores this string into a string variable.

Parameters

1. The first parameter specifies up to three items:
   - The first item is an expression of integral type or real type and is the value to be converted into a string.
2. The second item is optional, and if supplied is the minimum width of the string the value is converted to. If this item is not supplied then the minimum width of the string depends on the type of the value being converted. If an integral value is being converted then the minumum width of the string is eigth (8) characters, and if a real value is being converted then the minimum with of the string is nine (9) characters. **NOTE:** The string is padded on the left with spaces, if it is shorter than the minimum width.
3. The third item is optional, if a real value is being converted, and is the number of digits that follow the decimal point in the converted string (the real value is converted into fixed point representation). If third item is not supplied, and a real value is being converted, then the real value is converted into a exponential representation.
4. The second parameter is a reference to the string variable that will store the converted string value.

Example

```
str(1034, s);will store   '   1034'   in s
str(1034:1, s);will store   '1034'      in s
str(1034:5, s);will store   ' 1034'     in s
str(7893.678, s);will store   ' 7.89E+03' in s
str(7893.678:8, s);will store   '7.89E+03'  in s
str(7893.678:8:3, s);will store   '7893.678'  in s
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.57 The textbackground procedure

Description

The **textbackground** procedure sets the current text background color.

Calling this procedure does not immediately change the background color of the screen. The background colors of characters already on the screen are not changed, what this procedure does is to change the background color of characters written to the screen after the procedure is called. The following table shows the values used to specify various background colors.

value	color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White

Parameter

The **textbackground** procedure's only parameter is an expression of integral type that specifies the new text background color.

Example

```
//*************************************************************
// This program uses the built-in procedure "textbackground" to
// change the background color. If you are using Windows try
// running this program and see what happens.
//*************************************************************
program bc(output);
(*$W44-*) //Suppress warning message about "constant defined but no used",
//since we already know we are not using all of the constants.
const
black = 0;
blue = 1;
green = 2;
```

```
cyan = 3;
red = 4;
magenta = 5;
brown = 6;
white = 7;
blinking = 8;
(*$W44+*)
begin
write('Normal background ');
textbackground(blue + blinking);
write('Blinking blue background ');
textbackground(black);
writeln('Normal background ');
end.
```

Portability

Operating Systems: Windows only
Standard Pascal: No

7.4.58 The textcolor procedure

Description

The **textcolor** procedure sets the current text foreground color.

Calling this procedure does not immediately change the foreground color of the screen. The foreground colors of characters already on the screen are not changed, what this procedure does is to change the foreground color of characters written to the screen after the procedure is called. The following table shows the values used to specify various foreground colors.

value	color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Grey
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Yellow
15	High-intensity white

Parameter

The **textcolor** procedure's only parameter is an expression of integral type that specifies the new text foreground color.

Example

```
//*************************************************************
// This program uses the built-in procedure textcolor to
// set the screen foreground color.
program fc(output);
(*$W44-*)
const
black = 0;
blue = 1;
green = 2;
cyan = 3;
red = 4;
magenta = 5;
brown = 6;
white = 7;
grey = 8;
light_blue = 9;
light_green = 10;
light_cyan = 11;
light_red = 12;
light_magenta = 13;
yellow = 14;
high_intensity_white = 15;

(*$W44+*)
begin
textcolor(white);
write('White text ');
textcolor(light_green);
write('Light green text ');
textcolor(white);
writeln('White text ');
end.
```

Portability

Operating Systems:`Windows only`
Standard Pascal:`No`

7.4.59 The traperrors procedure

Description

The **traperrors** procedure turns automatic error trapping on or off.

If automatic error trapping is on and a trappable error occurs then the program is terminated, and an error message may be printed on the console screen, displayed in a message box, and/or logged to a file, depending on the project setting when the program was compiled.

If automatic error trapping is off and a trappable error occurs then one or more records of type error are created and added to the built-in list variable errors. Each error record describes the error that occured (some in more detail than others). Also the built-in function getlasterror will return a non-zero value to indicate which category of error occured.

If you do not want the program to be terminated when a trappable error occurs, and prefer to handle the error yourself then turn automatic error trapping off (with **TrapErrors(false)**) before executing code that may cause a trappable error. Then call getlasterror to determine which category of error occured, if any.

You will probably want to turn automatic error trapping back on (with **TrapErrors(true)**) after executing the code that might cause a trappable error.

The (*$I*) compiler directive, also turns automatic error trapping off and on, however new programs should use the **traperrors** procedure instead. Use of the (*I*) compiler directive continues to be supported for compatibility reasons only.

Parameter

The **traperrors** procedure's only parameter is an expression of boolean type that specifies wether automatic error trapping is being turned on or off.

Example

```
The following simple program asks for the name of a file and
then deterines whether the file exists. It determines whether
the file exists by checking whether the file can be opened for
reading. So technically it is really testing whether the file
file exists AND also if the user has read access to the file.
Anyway the built-in procedure "trapperrors" is used to turn off
error trapping before the program attempts to open the
file (this prevents the normal error trapping procedure from terminating
the program if the file can not be opened). Then the built-in procedure
"getlasterror" is called to determine if an error occured. If an error
occured this program just assumes that the file does not exist.

program DoesFileExist(input, output);
var
n : filename;

function FileExists(name : filename) : boolean;
var
f : text;
begin
traperrors(false);
reset(f, name);
FileExists := (getlasterror = 0);
if FileExists then
close(f);
traperrors(true)
end;

begin
writeln('Enter filename:');
readln(n);
n := trim(n);
writeln('Does ''', n, ''' exist?:', FileExists(n))
end.
```

Portability

Operating Systems: All
Standard Pascal: No

7.4.60 The unpack procedure

Description

The **unpack** procedure assigns values to some or all of the elements of an unpacked array by copying all of the elements of a packed array into the unpacked array. The elements of both arrays must have the same type.

Parameter

- The first parameter is a reference to the packed array variable that contains the array elements to copy.
- The second parameter is a reference to the unpacked array variable that contains the array elements to be copied into.
- The third parameter is an expression of ordinal type that specifies the start element of the unpacked array (i.e. the element of the unpacked array to start copying to). All of the elements of the unpacked array from the start element to the end are available to be copied into. It is an error if the number of elements in the unpacked array available to be copied into is less then the number of elements in the packed array, since all the elements in the packed array are copied. **NOTE:** This parameter must be assignment compatible with the index type of the unpacked array.

Example

Suppose you have the following arrays

```
p : packed array[21..23] of integer;
u : array[51..60] of integer;
```

then

```
unpack(p, u, 51)
```

copies elements 21 to 23 of **p** into elements 51 to 53 of **u**.

and

```
unpack(p, u, 57)
```

copies elements 21 to 23 of **p** into elements 57 to 59 of **u**.

and

```
unpack(p, u, 59)
```

produces an error since only two elements (i.e. 59 and 60) of **u** are available to be copied into but all three elements of **p** must be copied.

Portability

Operating Systems: All
Standard Pascal: Yes

7.4.61 The val procedure

Description

The **val** procedure converts a string value into an integral real value and assigns the converted value into a variable.

Parameters

- The first parameter is an expression of string type or char type, and is the string value to be converted.
- The second parameter is a reference to a variable of integral type or real type that stores the converted value.
- The third parameter is a reference to a variable of integer type or word type that stores the position of the first conversion error (i.e. the first character in the string value to be converted that could not be converted), or zero if there were no conversion errors.

Example

Given integer variables "i" and "x" the following:

```
val('123EB', i, x)
```

will store 123 in "i" and 4 in "x".

Give a real variable "r" and an integer variable "x" the following:

```
val('123.45', r, x)
```

will store 123.45 in "r" and 0 in "x".

Portability

Operating Systems: All
Standard Pascal: No

7.4.62 The write procedure

Description

The **write** procedure writes values to the file associated with a file variable.

Parameters

1. The first parameter is optional, and if supplied is a reference to the file variable associated with the file to write to. If this parameter is not supplied then the **write** procedure writes values to the file associated with output.

2. The other parameters are called write parameters, and specify the values to be written to the file.

Example

```
write(true);          (* writes '    true' *)
write(true:2);        (* writes 'tr' *)
write(3567);          (* writes '    3567' *)
write(3567:1);        (* writes '3567' *)
write(1.2345678);     (* writes ' 1.23E+00' *)
write(1.2345678:8);   (* writes ' 1.2E+00' *)
write(1.2345678:8:3); (* writes '   1.235' *)
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.4.63 The writeln procedure

Description

The **writeln** procedure writes a line into a text file.

Parameter

1. The first parameter is optional, and if supplied is a reference to the file variable associated with the file to write to. If this parameter is not supplied then the **writeln** procedure writes lines to the file associated with output.
2. The other parameters are optional, and if supplied are write parameters that specify the values in the line to be written to the file.

Example

```
writeln(true);          (* writes '    true' *)
writeln(true:2);        (* writes 'tr' *)
writeln(3567);          (* writes '    3567' *)
writeln(3567:1);        (* writes '3567' *)
writeln(1.2345678);     (* writes ' 1.23E+00' *)
writeln(1.2345678:8);   (* writes ' 1.2E+00' *)
writeln(1.2345678:8:3); (* writes '   1.235' *)
```

Portability

Operating Systems: All
Standard Pascal: Yes

7.5.1 Procedure declarations

Description

Procedure identifiers are identifiers that have been associated with a procedure using a procedure declaration.

Example

Here is an example of a procedure declaration.

```
(*******************************************************************
** PURPOSE: Writes the contents of one file into another.
** ARGUMENTS:
**    'f' - the input file
**    'g' - the output file
** NOTES: It is up to the caller to open and close the files.
*)
procedure WriteFile(var f, g : text);
var
c : char;
begin
while not eof(f) do
begin
read(f, c);
write(g, c)
end
end;
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
procedure-declaration =
procedure-heading ';'directive |
procedure-identification ';'procedure-block |
procedure-heading ';'procedure-block

block =declarative-part statement-part

compound-statement ='begin'statement-sequence 'end'

declaration-group =
label-declaration-group |
constant-definition-group |
type-definition-group |
variable-declaration-group |
function-declaration |
procedure-declaration

declarative-part ={declaration-group }

directive =forward-directive |external-directive

formal-parameter-list ='('formal-parameter-section {';'formal-parameter-section }')'

formal-parameter-section =value-parameter-specification |
variable-parameter-specification |
procedure-parameter-specification |
function-parameter-specification

function-heading ='function'identifier [formal-parameter-list ]':'result-type

function-identification ='function'function-identifier

function-identifier =identifier

function-parameter-specification =function-heading

identifier =letter {letter |digit }

identifier-list =identifier {','identifier }

procedure-block =block

procedure-heading ='procedure'identifier [formal-parameter-list ]

procedure-identification ='procedure'procedure-identifier

procedure-identifier =identifier

procedure-parameter-specification =procedure-heading

statement-part =compound-statement

statement-sequence =statement {';'statement }

type-identifier =identifier

value-parameter-specification =identifier-list ':'type-identifier

variable-parameter-specification ='var'identifier-list ':'type-identifier
```

7.6.1 Calling external functions and procedures

As an extension to Standard Pascal, Irie Pascal supports calling functions and procedures in Windows DLLs, using the external directive. **NOTE:** Only 32-bit DLLs are currently supported.

A number of include files, with declarations for many of the Windows API functions, are distributed with Irie Pascal. You can use these include files to gain access to many pre-written declarations of Windows API functions without having to create your own declarations for these functions.

In order to write your own declarations for external functions and procedures, you need to understand a little bit about the calling convention used by Windows DLLs. Windows DLLs support two calling conventions, **stdcall** and **cdecl**, the one most often used is **stdcall**. By default Irie Pascal will assume that the calling convention is **stdcall** unless **cdecl** is specified. Unfortunately the two calling conventions are not implemented in exactly the same way by all Windows compilers. So if you want to call functions and procedures in a DLL, you may need to know which compiler was used to create it (and also what compiler options were used).

This document does not describe the various ways different compilers implement the **stdcall** and **cdecl** calling conventions. Instead what this document does is describe how Irie Pascal implements these conventions.

- Irie Pascal passes parameters from right to left for both the **stdcall** and **cdecl** calling conventions.
- Irie Pascal expects the function or procedure being called to pop the parameters from the stack after the call, when the **stdcall** calling convention is being used, but the caller pops the parameters from the stack after the call, when the **cdecl** calling convention is being used.
- For both calling conventions, Irie Pascal searches for the function or procedure in the DLL using the name given, but if the function or procedure is not found then **"_"** is prefixed to the name and

the DLL is searched again. **NOTE:** This is done because some compilers automatically prefix function and procedure names with **"_"** but other compilers don't.

- For both calling conventions, Irie Pascal expects function results of ordinal type, pointer type, or address type to be returned in the EAX register.
- For both calling conventions, Irie Pascal expects function results of real type, single type, or double type to be returned on the math coprocessor stack.
- For both calling conventions, Irie Pascal expects function results of record type or array type to be returned using a pointer passed by the caller. The caller is expected to pass a pointer to the memory location where the function should put the return value.
- For both calling conventions, Irie Pascal does **not** support returning values of the following types: (dir, file, list, object, set, or string) from external functions and procedures.
- For both calling conventions, Irie Pascal supports passing expressions of the following types by value (ordinal types, pointer types, record types, address, double, single, and real).
- For both calling conventions, Irie Pascal supports passing variables of the following types by reference (array types, ordinal types, pointer types, record types, string types, address, double, single, and real).
- Irie Pascal does not yet support passing functions or procedures using either calling convention.

In order to successfully call external functions and procedures you may need to understand a little bit about how values of the Irie Pascal types are stored. Below is a brief description of the what values of the Irie Pascal types are stored:

First the simple types:

- **byte** - unsigned 8-bit value.
- **char** - unigned 8-bit value.
- **double** - 64-bit floating point value (see ANSI/IEC std 754-1985).
- **enumerated (including boolean)** - signed 32-bit value.
- **integer** - signed 32-bit value.
- **real** - 64-bit floating point value (see ANSI/IEC std 754-1985).
- **shortint** - signed 16-bit value.
- **shortword** - unsigned 16-bit value.
- **single** - 32-bit floating point value (see ANSI/IEC std 754-1985).
- **word** - unsigned 32-bit value.
- **subrange** - subrange types are stored like their host types (e.g. subranges of char are stored like chars, and subranges of integer are stored like integers).

Next the complex types:

- array - These are contiguous sequences of the elements. For example

  ```
  array[1..4] of integer
  ```

  is simply a sequence of 4 integers (with no gaps in between). Multi-demensional arrays are treated as follows:

  ```
  array[m][n][o][p] of type
  ```

  is treated like

  ```
  array[m] of array[n] of array[o] of array[p] of type.
  ```

  NOTE: This is the opposite of the way dimensions are ordered by C language compilers.
- **address** - This is a 32-bit value which is an operating system address (i.e. an address that is meaningful to external functions and procedures).
- **cstring** - These are null terminated sequences of characters (the same format used by C programs.

to store strings.

- **pointer** - This is a 32-bit value which is a virtual address (i.e. an offset from the beginning of the IVM Executable's Data segment). Virtual address are probably meanless to external functions and procedures and therefore you should use the built-in function addr to convert virtual addresses into operating system addresses before passing them to external functions or procedures.
- **record** - These are simply sequences of the fields. The size of the gaps (if any) between fields is determined by the size of the fields and the current alignment setting.
- **string** - These are length prefixed sequences of characters. Strings with maximum lengths less than or equal to 255 are prefixed by a byte containing the actual length of the string. String with maximun lengths greater than or equal to 256 are prefixed by an integer containing the actual length of the string. **NOTE:** This representation is typical of the way strings are represented in Pascal programs, but is very different from the way way strings are repesented in C programs (i.e. null terminated sequences of char).
- The other types (such as **file** and **list**) are not described here because they can not be passed to or from external functions or procedures.

Most functions in Windows DLLs are written in C, so you will usually be converting back of forth between C types and Irie Pascal types.

NOTE: The C types in all capitals like BOOL, WORD, and DWORD are not actually defined as part of the C language, but are common types declared in C programs.

Using BOOL Parameters

The C languages does not define a boolean type so it is common for C programs to define one, however there are two popular ways this can be done. One way is to define the boolean type as **char** or **unsigned char**, in which case the nearest Irie Pascal type is **byte**. The other common way to define the boolean type is as a enumerated type with the values **FALSE**, and **TRUE**. However sometimes C compilers treat enumerated types with less than 256 elements as **unsigned char**, and at other times as **int** (the C name for the integer type). So if boolean type is defined as an enumerated type then it can either be treated as **unsigned char**, in which case the nearest Irie Pascal type is **byte**, or as **int**, in which case the nearest Irie Pascal type is **boolean**.

Using int And long Parameters

With most C compilers, the types **int** and **long** are signed 32-bit integral types so the nearest Irie Pascal type is **integer**.

Using DWORD, unsigned int And unsigned long Parameters

With most C compilers, the types **DWORD**, **unsigned int**, and **unsigned long** are unsigned 32-bit integral types so the nearest Irie Pascal type is **word**.

Using short Parameters

With most C compilers, the type **short** is a signed 16-bit integral type so the nearest Irie Pascal type is **shortint**.

Using unsigned short Parameters

With most C compilers, the type **unsigned short** is a unsigned 16-bit integral type so the nearest Irie

Pascal type is **shortword**.

Using Array Parameters

Passing multi-dimensional arrays between Irie Pascal and C requires care since the ordering used by Pascal and C for the dimensions is opposite (i.e.

```
a : array[1..4][1..2] of integer;
```

is equivalent to

```
int a[2][4];
```

Using pointer Parameters

1. The C language does not support call by reference so it is common when passing a variable to a function to be modified, to pass a pointer to the variable. The function can't modify the pointer but it can modify what the pointer is pointing at (i.e. the variable). Pascal supports call by reference, so you may want to use it in cases where the C function or procedure expects a pointer to a variable.
2. The C language also does not support passing arrays to functions, however the close relationship between arrays and pointers in C means that passing a pointer to the first element in an array is almost equivalent to passing the array. Irie Pascal supports passing arrays, so in this case you should pass an array by reference to the C function or procedure.
3. A special case of passing arrays, is passing C strings (arrays of null-terminated chars). In this case you may want to pass a cstring by reference.
4. In cases where the pointer is pointing to a variable whose type is not fixed, or a null value can be passed, or the pointer is pointing to a variable whose size is not fixed then you may want to pass the operating system address of the variable directly.

When you want to call a C function that expects a pointer parameter it is up to you to determine (perhaps by reading a description of the function) which one of the four scenarios just described in being used.

The sample program **winhello.pas** is an example of how to call a function in a Windows DLL.
NOTE: This program uses the **winuser.inc** include file, distributed with Irie Pascal, to gain access to a pre-written declaration for the Windows API function **MessageBox**.

8.1 What are expressions?

Description

Expressions specify that zero or more operations are to be performed on a group of operands (also known as factors), in such a way as to result in a single value of a particular type. If the number of operations specified by an expression is zero, then the expression must consist of a single operand, and the value of the expression is the value of the operand. The process of obtaining the value of an expression, by obtaining the values of it's operands and performing any specified operations, is called evaluating the expression. The operations to be performed by an expression are specified using operators.

Example

Here are some examples of expressions:

```
3+5which results in the integer value 8
truewhich results in the boolean value true
3+5.0which results in the real value 8.0
1+2*3which results in the integer value 7
(1+2)*3which results in the integer value 9
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
expression =shift-expression [relational-operator shift-expression ]

adding-operator ='+'|'-'|'or'|'or_else'|'xor'

factor =[sign ]unsigned-constant |
[sign ]variable-access |
[sign ]'('expression ')'|
[sign ]function-designator |
[sign ]function-method-designator |
[sign ]'not'factor |
set-constructor

multiplying-operator ='*'|'/'|'div'|'mod'|'and'|'and_then'

relational-operator ='='|'<>'|'<'|'<='|'>'|'>='|'in'

shift-expression =simple-expression [shift-operator simple-expression ]

shift-operator ='shl'|'shr'

sign ='-'|'+'

simple-expression =term {adding-operator term }

term =factor {multiplying-operator factor }
```

8.2 Set constructors

Description

Set constructors are used in expressions to create values of set types, and contain comma-delimited lists of zero or more member designators. Member designators specify set elements which are present, and contain a single ordinal expression, or a pair of ordinal expressions. If the member designator contains a single ordinal expression, them the value of this expression is the set element that is present. If the member designator contains a pair of ordinal expressions, then the member designator specifies a range of set elements that are present, and the value of the first ordinal expression is the first set element in the range, and the value of the second ordinal expression is the last set element in the range.

A set constructor with no member designators is valid and denotes the empty set (i.e. no set elements are present).

Example

Here are some examples of set constructors:

```
[]- the empty set
[true]- set of boolean with only truepresent
['a'..'z','A'..'Z']- set of char with all the lowercase and uppercase letters present
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
set-constructor ='['[member-designator {','member-designator }]']'

member-designator =ordinal-expression ['..'ordinal-expression ]

ordinal-expression =expression
```

8.3.1 What are type conversion rules

Description

Type conversion is the process of converting an operand of one type into an operand of another type, and is usually done to convert one operand to the same type as another operand, before performing an operation involving both operands. For example, if a value of integer type is to be added to a value of real type, then the value of integer type is first converted to a value of real type before the addition takes place.

Irie Pascal automatically performs type conversions on the operands of expressions where necessary, by applying one of the following rules:

- The numeric type conversion rules.
- The char/string type conversion rules.

8.3.2 Numeric type conversion rules

Description

Irie Pascal applies the numeric type conversion rules when performing an operation with two numeric operands, to convert one or both operands to the same type.

- Any operand of a subrange type is first converted to its host type.
- Any operand of byte type, or shortint type, or shortword type is converted to integer type.
- If one operand is of integer type and the other operand is of type word then the operand of type integer is converted to type word.
- If one operand is of real type, and the other operand is not, then the other operand is converted to real type.

8.3.3 Char/String type conversion rules

Description

Irie Pascal applies the char/string conversion rules when performing an operation with two operands of char type or string type, to convert one or both operands to the same type.

- If one operand is of char type and the other operand is of string type then the operand of type char is converted to type string.

8.4.1 What are operators?

Below is a table of all the operators supported by Irie Pascal in order of decreasing precedence. All operators on the same line have the same precedence. Operations with operators of the same precedence are performed left to right. Parantheses can be used to control the order in which operations are performed by grouping operations that should be performed first.

```
-----------------------------
not
-----------------------------
*  /  div  mod  and  and_then
-----------------------------
+  -  or  or_else  xor
-----------------------------
shl  shr
-----------------------------
=  <> <  <=  >  >=  in
-----------------------------
```

So for example with the expression below

```
2 + 3 - 1
```

the addition would be performed before the subtraction. This is because "+" and "-" have the same precedence and so the operations are performed from left to right.

However since "*" has a higher precedence then with the expression below

```
2 + 3 * 1
```

the multiplication is permormed before the addition. Using the same example above you can force the addition to be performed first by using parentheses like so:

```
(2 + 3) * 1
```

For more complicated expressions you can use nested parentheses, in which case the operatings enclosed in the inner-most parentheses are performed first. So for example given

```
2 + (4 - 1) * (4 div (2 + 2))
```

the first operation performed is 2 + 2 which yields 4, resulting in the expression below

```
2 + (4 - 1) * (4 div 4)
```

Now we have two nested parentheses at the same level so which operation is performed first? Well operations at the same level of precedence are performed left to right so the (4 - 1) is performed next resulting in

```
2 + 3 * (4 div 4)
```

Next the 4 div 4 is performed resulting in

```
2 + 3 * 1
```

Next the 3 * 1 is performed resulting in

```
2 + 3
```

which results in the integer value 5

Operator Categories

Operators can be grouped into the following categories:

- Arithmetic operators
- Equality operators
- Relational operators
- Boolean operators
- String operators
- Set operators
- Bitwise operators

8.4.2.1 The arithmetic operators

The arithmetic operators are:

- Addition (+)
- Unary plus (+)
- Subtraction (-)
- Unary minus (-)
- Multiplication (*)
- Real division (/)
- integer division (div)
- Modulus (mod)

8.4.2.2 Addition (+)

Description

When the + operator has two operands of integral type or real type it performs addition, according to the following rules:

First the numeric type conversion rules are applied. After applying the numeric type conversion rules, the operands are either both of integer type, or both of word type or both of real type, or one operand is of address type and the other operand is not.

- If both operands are of integer type then unsigned integer addition is performed, and the result is a value of type integer.

- If both operands are of word type then unsigned integer addition is performed, and the result is a value of type word.
- If both operands are of real type then real (i.e floating point) addition is performed, and the result is a value of type real.
- If one operand is of address type and the other operand is not then unsigned integer addition is performed, and the result is a value of address type.

Example

Here are some examples using the + operator to perform addition:

```
3 + 5which results in the integer value 8
3 + 5.0which results in the real value 8.0
3.0 + 5which results in the real value 8.0
3.0 + 5.0which results in the real value 8.0
```

8.4.2.3 Unary plus (+)

Description

When the + operator has only one operand of integral type or real type it performs unary plus which means that the result of the operation is the operand (in other words unary plus does nothing).

8.4.2.4 Subtraction (-)

Description

When the **-** operator has two operands of integral type or real type it performs subtraction, according to the following rules:

First the numeric type conversion rules are applied. After applying the numeric type conversion rules, the operands are either both of integer type, or both of word type or both of real type, or both of address type or one operand is of address type and the other operand is not.

- If both operands are of integer type then unsigned integer subtraction is performed, and the result is a value of type integer.
- If both operands are of word type then unsigned integer subtraction is performed, and the result is a value of type word.
- If both operands are of real type then real (i.e floating point) subtraction is performed, and the result is a value of type real.
- If both operands are of address type then unsigned integer subtraction is performed, and the result is a value of type integer.
- If one operand is of address type and the other operand is not then unsigned integer subtraction is performed, and the result is a value of address type.

Example

Here are some examples using the **-** operator to perform subtraction:

```
3 - 5which results in the integer value -2
3 - 5.0which results in the real value -2.0
3.0 - 5which results in the real value -2.0
3.0 - 5.0which results in the real value -2.0
```

8.4.2.5 Unary minus (-)

Description

When the **-** operator has only one operand of integral type or real type it performs unary minus (which is the same as subtracting the operand from zero).

8.4.2.6 multiplication (*)

Description

When the * operator has two operands of integral type or real type it performs multiplication, according to the following rules:

First the numeric type conversion rules are applied. After applying the numeric type conversion rules, the operands are either both of integer type, or both of word type or both of real type.

- If both operands are of integer type then integer multiplication is performed, and the result is a value of integer type.
- If both operands are of word type then integer multiplication is performed, and the result is a value of word type.
- If both operands are of real type then real multiplication is performed, and the result is a value of real type.

Example

Here are some examples using the * operator to perform multiplication:

```
3 * 5which results in the integer value 15
3 * 5.0which results in the real value 15.0
3.0 * 5which results in the real value 15.0
3.0 * 5.0which results in the real value 15.0
-3 * 5which results in the integer value -15
3 * -5which results in the integer value -15
-3 * -5which results in the integer value 15
```

8.4.2.7 Real division (/)

Description

When the / operator has two operands of integral type or real type it performs real division and the result is a value of real type. Any operands of integral type are converted into operands of real type before the division is performed.

Example

Here are some examples using the / operator to perform division:

```
5 / 2which results in the real value 2.5
5.0 / 2.0which results in the real value 2.5
-5 / 2which results in the real value -2.5
5.0 / -2.0which results in the real value -2.5
-5.0 / -2.0which results in the real value 2.5
```

8.4.2.8 Integer division (div)

Description

The **div** operator takes two operands of integral type, performs integral division, and the result is a value of integral type.

Example

Here are some examples using the **div** operator to perform division:

```
5 div 2which results in the real value 2
-5 div 2which results in the real value -2
5 div -2which results in the real value -2
-5 div -2which results in the real value 2
```

8.4.2.9 Modulus (mod)

Description

The **mod** operator takes two operands of integral type and calculates the modulus. The result of this operation is of integral type. The modulus operation is defined by Standard Pascal (ISO/IEC 7185) as follows:

*A term of the form **i mod j** shall be an error if **j** is zero or negative, otherwise, the value of **i mod j** shall be that value of **(i-(k*j))** for integral **k** such that **0 <= i mod j < j***

Irie Pascal implements the **mod** operator according to the definition given above. What does the definition really mean? Well in the simple case where the operands are positive then the result of the operation is the remainder left over after dividing the left operand by the right operand. This is the most common use of the **mod** operator, but you should be aware that this does not hold when the left operand is negative. You should also be aware that the right operand can not be negative.

Example

For example if you run the program below:

```
program mods(output);
var
i : -3..3;
j : 1..3;
begin
for i := -3 to 3 do
for j := 1 to 3 do
writeln(i:1, ' mod ', j:1, ' = ', i mod j : 1);
end.
```

you will get the following output

```
-3 mod 1 = 0
-3 mod 2 = 1
-3 mod 3 = 0
-2 mod 1 = 0
-2 mod 2 = 0
-2 mod 3 = 1
-1 mod 1 = 0
-1 mod 2 = 1
-1 mod 3 = 2
0 mod 1 = 0
0 mod 2 = 0
0 mod 3 = 0
1 mod 1 = 0
1 mod 2 = 1
1 mod 3 = 1
2 mod 1 = 0
2 mod 2 = 0
2 mod 3 = 2
3 mod 1 = 0
3 mod 2 = 1
3 mod 3 = 0
```

8.4.3.1 The equality operators

The equality operators are:

- Is equal (=)
- Is not equal (<>)

8.4.3.2 Is equal

Description

The = operator takes two operands and compares them for equality. If the operands are numeric then the numeric type conversion rules are applied before the comparision is done. If the operands are of char type or of string type then the char/string type conversion rules are applied before the comparision is done. The result of this operation is of boolean type, and is true if the operands are equal, and false otherwise.

Example

```
23 =   23    results in true
-23 =  -23    results in true
23 = 23.0    results in true
23.0 =   23    results in true
23 =  -23    results in false
2 =    3    results in false
3 =    2    results in false
```

8.4.3.3 Is not equal

Description

The <> operator takes two operands and compares them for equality. If the operands are numeric then the numeric type conversion rules are applied before the comparision is done. If the operands are of char type or of string type then the char/string type conversion rules are applied before the comparision is done. The result of this operation is of boolean type, and is true if the operands are **not** equal, and false otherwise.

Example

```
23 <>   23    results in false
-23 <>  -23    results in false
23 <> 23.0    results in false
23.0 <>   23    results in false
23 <>  -23    results in true
2 <>    3    results in true
3 <>    2    results in true
```

8.4.4.1 The relational operators

The relational operators are:

- Is less than
- Is less than or equal
- Is greater than
- Is greater than or equal

With numeric operands the relational operators compare the values of the operands.

For operands of string type the relational operators compare the lexical ordering of the operands.

For operands of set type the relational operators (<= and >= only) test whether one operand is a subset of the other.

8.4.4.2 Is less than

Description

The < operator takes two operands and compares them. If the operands are numeric then the numeric type conversion rules are applied before the comparision is done. If the operands are of char type or of string type then the char/string type conversion rules are applied before the comparision is done. The result of this operation is of boolean type and is true if the left operand is less than the right operand, and false otherwise.

Example

For example

```
23 <   23    results in false
-23 <  -23    results in false
23 < 23.0    results in false
23.0 <   23    results in false
2 <    3    results in true
2 <   -3    results in false
-2 <   -3    results in false
3 <    2    results in false
-3 <    2    results in true
-3 <   -2    results in true
```

and

```
'abc' < 'abc'  results in false
'abc' < 'abcd' results in true
'abc' < 'ab'   results in false
'abc' < 'abd'  results in true
'abc' < 'aac'  results in false
'abc' < 'b'    results in true
```

8.4.4.3 Is less than or equal

Description

When the <= operator has two numeric operands, it compares their values, after applying the numeric type conversion rules. The result of this operation is of boolean type and is true if the left operand is less than or equal to the right operand, and false otherwise.

When the <= operator has two operands of char type or of string type then the char/string type conversion rules are applied, and then the lexical ordering of the operands are compared. The result of this operation is of boolean type and is true if the left operand is less than or equal to the right operand, and false otherwise.

When the <= operator has two operands of set type then it tests for set inclusion. The result of this operation is true if all the members of the left operand are in the right operand, and false otherwise.

Example

For example:

```
23 <=   23    results in true
-23 <=  -23    results in true
23 <= 23.0    results in true
23.0 <=   23    results in true
2 <=    3    results in true
2 <=   -3    results in false
-2 <=   -3    results in false
3 <=    2    results in false
-3 <=    2    results in true
-3 <=   -2    results in true
```

and

```
'abc' <= 'abc'  results in true
'abc' <= 'abc' results in true
'abc' <= 'ab'  results in false
'abc' <= 'abd'  results in true
'abc' <= 'aac'  results in false
'abc' <= 'b'    results in true
```

and

```
['a', 'b', 'c'] <= ['a']            returns false
['a', 'b', 'c'] <= ['a', 'b']       returns false
['a', 'b', 'c'] <= ['a', 'b', 'c'] returns true
['a']           <= ['a', 'b', 'c'] returns true
['a', 'b']      <= ['a', 'b', 'c'] returns true
[]              <= ['a']            return true
['a']           <= []               return false
```

8.4.4.4 Is greater than

Description

When the > operator has two numeric operands, it compares their values, after applying the numeric type conversion rules. The result of this operation is of boolean type and is true if the left operand is greater than the right operand, and false otherwise.

When the > operator has two operands of char type or of string typethen the char/string type conversion rules are applied, and then the lexical ordering of the operands are compared. The result of this operation is of boolean type and is true if the left operand is greater than the right operand, and false otherwise.

Example

For example

```
'abc' > 'abc'  results in false
'abc' > 'abcd' results in false
'abc' > 'ab'   results in true
'abc' > 'abd'  results in false
'abc' > 'aac'  results in true
'abc' > 'b'    results in false
```

8.4.4.5 Is greater than or equal

Description

When the >= operator has two numeric operands, it compares their values, after applying the numeric type conversion rules. The result of this operation is of boolean type and is true if the left operand is greater than or equal to the right operand, and false otherwise.

When the >= operator has two operands of char type or of string typethen the char/string type conversion rules are applied, and then the lexical ordering of the operands are compared. The result of this operation is of boolean type and is true if the left operand is greater than or equal to the right operand, and false otherwise.

When the >= operator has two operands of set type then it tests for set inclusion. The result of this operation is true if all the members of the right operand are in the left operand, and false otherwise.

Example

For example:

```
23 >=   23    results in true
-23 >=  -23    results in true
23 >= 23.0    results in true
23.0 >=   23    results in true
2 >=    3    results in false
2 >=   -3    results in true
-2 >=   -3    results in true
3 >=    2    results in true
-3 >=    2    results in false
-3 >=   -2    results in false
```

and

```
'abc' >= 'abc'  results in true
'abc' >= 'abcd' results in false
'abc' >= 'ab'   results in true
'abc' >= 'abd'  results in false
'abc' >= 'aac'  results in true
'abc' >= 'b'    results in false
```

and

```
['a', 'b', 'c'] >= ['a']            returns true
['a', 'b', 'c'] >= ['a', 'b']       returns true
['a', 'b', 'c'] >= ['a', 'b', 'c']  returns true
['a']           >= ['a', 'b', 'c']  returns false
['a', 'b']      >= ['a', 'b', 'c']  returns false
[]              >= ['a']            return false
['a']           >= []               return true
```

8.4.5.1 The boolean operators

The boolean operators are:

- not (boolean)

8.4.5.2 not (boolean)

Description

When the **not** operator has one operand of boolean type then the result of the operation is true if the value of the operand is false, and the result of the operation is false if the value of the operand is true.

Example

```
not true   results in "false"
not false  results in "true"
```

8.4.5.3 and (boolean)

Description

When the **and** operator has operands of type boolean type then it performs boolean AND. The result of the operation is of boolean type and is equal to true of both operands are true, and false if either operand is false. By default short circuit evaluation is used when performing the boolean AND (i.e. The left operand is evaluated first and if it is false then the right operand is not evaluated because the result of the operation must be false). A compiler option can be used to enable/disable short-circuit evaluation.

Example

```
false and false   results in false
false and true    results in false
true  and false   results in false
true  and true    results in true
```

8.4.5.4 and_then (boolean)

Description

As an extension to Standard Pascal, Irie Pascal supports the **and_then** operator, which takes two operands of boolean type and performs boolean AND. The result of the operation is of boolean type and is equal to true of both operands are true, and false if either operand is false. Short circuit evaluation is always used by the **and_then** operator (i.e. The left operand is evaluated first and if it is false then the right operand is not evaluated because the result of the operation must be false).

Example

```
false and_then false   results in false
false and_then true    results in false
true  and_then false   results in false
true  and_then true    results in true
```

The differences between the **and_then** operator and the **and** operator are:

- The **and_then** operator can not be used to perform bitwise AND.
- Short-circuit evaluation is always used when by the **and_then** operator but can be enabled/disabled for the **and** operator.

8.4.5.5 or (boolean)

Description

When the **or** operator has operands of boolean type then it performs boolean OR. The result of the operation is of boolean type and is equal to true if either operand is true, and false if both operands are false. By default short-circuit evaluation is used when performing the boolean OR (i.e. The left operand is evaluated first and if it is true then the right operand is not evaluated because the result of the operation must be true). A compiler option can be used to enable/disable short-circuit evaluation.

Example

```
false or false   results in false
false or true    results in true
true  or false   results in true
true  or true    results in true
```

8.4.5.6 or_else (boolean)

Description

As an extension to Standard Pascal, Irie Pascal supports the **or_else** operator, which takes two operands of boolean type and performs boolean OR. The result of the operation is of boolean type, and is equal to true if either operand is true, and false if both operands are false. Short-circuit evaluation is always used by the **or_else** operator (i.e. The left operand is evaluated first and if it is true then the right operand is not evaluated because the result of the operation must be true).

Example

```
false or_else false   results in false
false or_else true    results in true
true  or_else false   results in true
true  or_else true    results in true
```

The differences between the **or_else** operator and the **or** operator are:

- The **or_else** operator can not be used to perform bitwise OR.
- Short-circuit evaluation is always used when by the **or_else** operator but can be enabled/disabled for the **or** operator.

8.4.5.7 xor (boolean)

Description

As an extension to Standard Pascal, Irie Pascal supports the **xor** operator, which when used with operands of boolean type performs boolean XOR. The result of the operation is of boolean type, and is equal to true if one but not both operands are true, and false if both operands are false, or both operands are true.

Example

```
false xor false   results in false
false xor true    results in true
true  xor false   results in true
true  xor true    results in false
```

8.4.6.1 The string operators

The only string operator is string concatenation (+).

8.4.6.2 String concatenation (+)

Description

As an extension to Standard Pascal, Irie Pascal supports using the + operator to perform string concatenation (i.e. joining). When the + operator has two operands of string type or char type then it performs string concatenation. Any operands of char type are converted to string type before being concatenated.

Example

```
'Hello' + ' ' + 'world' + '!!'   results in 'Hello world!!'
```

8.4.7.1 The set operators

The set operators are:

- Set union (+)
- Set difference (-)
- Set intersection (*)
- in

8.4.7.2 Set union (+)

Description

When the + operator has operands of set type then it performs set union (i.e. the result of the operation is a set with all the members from both operands).

Example

```
['a', 'e'] + ['a', 'c', 'z'] results in ['a', 'c', 'e', 'z']
```

To see this try the example program below:

```
program setu(output);
type
letter = 'a'..'z';
var
s : set of letter;
c : letter;
begin
s := ['a', 'e'] + ['a', 'c', 'z'];
for c := 'a' to 'z' do
if c in s then
writeln(c)
end.
```

8.4.7.3 Set difference (-)

Description

When the **-** operator has operands of set type then it performs set difference (i.e. the result of the operation is a set made up of the members in the left operand that are not in the right operand).

Example

```
['a', 'b', 'c', 'd'] - ['a', 'j'] results in ['b', 'b', 'c', 'd']
```

To see this try the example program below:

```
program setd(output);
type
letter = 'a'..'z';
var
s : set of letter;
c : letter;
begin
s := ['a', 'b', 'c', 'd'] - ['a', 'j'];
for c := 'a' to 'z' do
if c in s then
writeln(c)
end.
```

8.4.7.4 Set intersection (*)

Description

When the * operator has operands of set type then it performs set intersection (i.e. the result of the operation is a set made up of the members common to both the left and right operands).

Example

```
['a', 'b', 'c', 'd'] * ['a', 'c', 'j', 'z']  results in ['a', 'c']
```

To see this try the example program below:

```
program seti(output);
type
letter = 'a'..'z';
var
s : set of letter;
c : letter;
begin
s := ['a', 'b', 'c', 'd'] * ['a', 'c', 'j', 'z'];
for c := 'a' to 'z' do
if c in s then
writeln(c)
end.
```

8.4.7.5 Set inclusion (in)

Description

The **in** operator takes two operands, the left operand is of ordinal type and the right operand is of set type. The result of the operation is of boolean type and is true if the left operand is a member of the right operand, and is false otherwise.

Example

```
'a'    in  ['a', 'b', 'c']    returns true
'b'    in  ['a', 'b', 'c']    returns true
'c'    in  ['a', 'b', 'c']    returns true
'd'    in  ['a', 'b', 'c']    returns false
'e'    in  ['a', 'b', 'c']    returns false
'a'    in  []                 returns false
```

8.4.8.1 The bitwise operators

The bitwise operators are:

- not (bitwise)
- and (bitwise)
- or (bitwise)
- xor (bitwise)

- Bit shift left (shl)
- Bit shift right (shr)

8.4.8.2 not (bitwise)

Description

As an extension to Standard Pascal, Irie Pascal supports using the **not** operator with an operand of integral type to perform bitwise NOT. The result of the operation is calculated by flipping all the bits in the operand.

Example

```
not 1      //results in -2
//Since not 1 = not %00000000000000000000000000000001
// which results in %11111111111111111111111111111110
// or -2
```

8.4.8.3 and (bitwise)

Description

As an extension to Standard Pascal, Irie Pascal supports using the **and** operator with two operands of integral type to perform bitwise AND. The result of the operation is of integral type, and has its bits set as follows: each bit is one (1) in the result if and only if the both of the bits in the corresponding position in the operands are one (1).

Example

```
%11100 and %10111  results in %10100
```

Binary notation is used in the example above so that you can clearly see the individual bits in the operands and the result. The example above could have been written as

```
28 and 23   results in  20
```

which is the same thing, but is probably not as clear.

8.4.8.4 or (bitwise)

Description

As an extension to Standard Pascal, Irie Pascal supports using the **or** operator with two operands of integral type to perform bitwise OR. The result of the operation is of integral type, and has its bits set as follows: each bit is one (1) in the result if and only if one of the bits in the corresponding position in the operands is one (1).

Example

```
%11100 or %10111  results in %11111
```

Binary notation is used in the example above so that you can clearly see the individual bits in the operands and the result. The example above could have been written as

```
28 or 23   results in  31
```

which is the same thing, but is probably not as clear.

8.4.8.5 xor (bitwise)

Description

When the **xor** operator has operands of integral type then it performs bitwise exclusive or. The result of the operation is of integral type and has its bits set as follows: each bit is one (1) in the result if and only if one of bits in the corresponding position in the operands is one (1) but not both.

Example

```
%11100 xor %10111  results in %01011
```

Binary notation is used in the example above so that you can clearly see the individual bits in the operands and the result. The example above could have been written as

```
28 xor 23   results in  11
```

which is the same thing, but is probably not as clear.

8.4.8.6 Bit shift left (shl)

Description

As an extension to Standard Pascal, Irie Pascal supports the **shl** operator, which takes two operands of integral type and performs a left bit-shift. The result of the operation is of integral type, and is calculated by shifting to the left, the bits in the left operand, by the number of places specified by the right operand. The right operand must be greater than or equal to zero. **NOTE:** Bit-shifting left by N bits is the same as multiplying by 2 raised to the Nth power, for positive operands.

Example

```
%11100 shl 3  results in %11100000
```

Binary notation is used in the example above so that you can clearly see the individual bits in the operands and the result. The example above could have been written as

```
28 shl 3   results in 224
```

which is the same thing, but is probably not as clear.

8.4.8.7 Bit shift right (shr)

Description

As an extension to Standard Pascal, Irie Pascal supports the **shr** operator, which takes two operands (left and right) of integral type and performs a right bit-shift operation. The result of the operation is of integral type, and is calculated by shifting to the right, the bits in the left operand, by the number of places specified by the right operand. The right operand must be greater than or equal to zero. **NOTE:** Bit-shifting right by N bits is the same as dividing by 2 raised to the Nth power, for positive operands.

Example

```
%11100 shr 3  results in %00011
```

Binary notation is used in the example above so that you can clearly see the individual bits in the operands and the result. The example above could have been written as

```
28 shr 3   results in 3
```

which is the same thing, but is probably not as clear.

9.1 What are statements?

The statements in a program are the executable part of the program (i.e. when executed they perform the program's *actions*). Currently, all programs created by Irie Pascal have a single thread of execution, so only one statement can be executed at a time. Usually, the statements in a program are executed in the order in which they occur in the text of the program However some statements can change the order in which the statements in a program are executed, cause some statements not to be executed at all, or cause some statements to be executed more than once.

Pascal programs always begin executing at the first statement in the main program block.

Statements can be grouped in a number of ways. For example statements can be separated into two groups, simple statement and structured statements, where simple statements are statements that do not contain other statements, while structured statements are statements that do contain other statements. Statements can also be grouped according to the kind of action they perform, such as conditional statements and repetitive statements. Conditional statements allow programs to make decisions (i.e. they test a particular condition and transfer program execution depending on the result). Repetitive statements allow the program to loop (i.e. transfer program execution to the same statement or statements repeatedly).

Labels can be placed in front of statements to mark them as targets for goto statements. As an extension to Standard Pascal, Irie Pascal allows labels to be identifiers as well as sequences of digits.

The statements are:

- Empty statement
- Assignment statement
- Procedure statement

- Goto statement
- Compound statement
- If statement
- Case statement
- Repeat statement
- While statement
- For statement
- With statement
- Procedure method call

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
statement =[label ':'](simple-statement |structured-statement )

actual-parameter =expression |variable-access |
procedure-identifier |function-identifier

actual-parameter-list ='('actual-parameter {','actual-parameter }')'

assignment-statement =assignment-statement-lhs ':='expression

assignment-statement-lhs =variable-access |function-identifier |property-designator

case-statement ='case'case-index case-body [';']'end'

compound-statement ='begin'statement-sequence 'end'

conditional-statement =if-statement |case-statement

else-part ='else'statement

empty-statement =

for-statement ='for'control-variable ':='initial-value ('to'|'downto')final-value
'do'statement

goto-statement ='goto'label

if-statement ='if'boolean-expression 'then'statement [else-part ]

label =digit-sequence |identifier

procedure-identifier =identifier

procedure-method-identifier =identifier

procedure-method-specifier =object-variable '.'procedure-method-identifier

procedure-method-statement =procedure-method-specifier [actual-parameter-list ]

procedure-statement =procedure-identifier (
[actual-parameter-list ]|
read-parameter-list |readln-parameter-list |
write-parameter-list |writeln-parameter-list
)

repeat-statement ='repeat'statement-sequence 'until'boolean-expression

repetitive-statement =repeat-statement |while-statement |for-statement

simple-statement =empty-statement |assignment-statement |
procedure-statement |procedure-method-statement |
goto-statement

statement-sequence =statement {';'statement }

structured-statement =compound-statement |
conditional-statement |
repetitive-statement |
with-statement

while-statement ='while'boolean-expression 'do'statement

with-statement ='with'record-variable-list 'do'statement
```

9.2 Empty statement

Description

Empty statements don't perform any action, and are usually created when the rules of Pascal require a statement to exist but you don't specify one. For example if you write

```
begin
end
```

then technically according to the rules of Pascal there must be a statement between the **begin** and the **end**, and since there is nothing there the empty statement is said to exist there. Empty statements also commonly occur when statement sequences have a trailing semi-colon. In Pascal, the semi-colon is used as a statement separator and not a statement terminator so the following statement-sequence

```
a;
b;
```

actually contains three statements, the statement **a**, the statement **b**, and an empty statement following **b**. Since a semi-colon follows **b** then it must separate **b** from another statement.

Syntax

```
empty-statement =
```

9.3 Assignment statement

Description

Assignments statements are made up of a left-hand side and a right-hand side, seperated by the assignment operator (**:=**). The left-hand side of an assignment statement is either a variable, function identifier, or a property of an instance variable (see object variables for information about instance variables). The right-hand side of an assignment statement is an expression.

When a assignment statement is executed, its right-hand side is evaluated and its value is stored in the left-hand side. The value of the right-hand side must be assignment compatible with the type of the left-hand side.

NOTE: Functions return values by assigning values to their function identifiers.

Example

```
x := 1+1
```

is a simple example. When this statement is executed the right-hand side

```
1+1
```

is evaluated, and the resulting value (i.e. **2**) is stored in the left-hand side (i.e. the variable **x**).

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
assignment-statement =assignment-statement-lhs ':='expression

assignment-statement-lhs =variable-access |function-identifier |property-designator

property-designator =object-variable '.'property-specifier
```

9.4 Procedure statement

Description

Procedure statements execute procedures (which may involve passing actual parameters to the procedures formal parameters). **Procedure statements** are executed as follows:

1. First, the actual parameters, if present, are passed to the formal parameters, of the procedure. If the procedure has no formal parameters then this step is skipped.
2. Next, program execution is transfered to the first statement in the procedure's block.
3. Execution of the **procedure statement** is terminated when either the execution of the last statement in the procedure's block is terminated, or when one of the statements in the procedure's block transfers program execution to a statement outside of the procedure's block.

Example

```
writeln('1 + 1 = ', 1+1)
```

is a simple example. When this procedure statement is executed the actual parameters

```
'1 + 1 = '
```

and

```
1+1
```

are passed to the procedure and then the procedure prints the actual parameters to the file associated with output.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
procedure-statement =procedure-identifier (
[actual-parameter-list ]|
read-parameter-list |readln-parameter-list |
write-parameter-list |writeln-parameter-list
)

actual-parameter =expression |variable-access |
procedure-identifier |function-identifier

actual-parameter-list ='('actual-parameter {','actual-parameter }')'

function-identifier =identifier

procedure-identifier =identifier

read-parameter-list ='('[file-variable ',']variable-access {','variable-access }')'

readln-parameter-list =['('(file-variable |variable-access ){','variable-access }')']

write-parameter =expression [':'expression [':'expression ]]

write-parameter-list ='('[file-variable ',']write-parameter {','write-parameter }')'

writeln-parameter-list =['('(file-variable |write-parameter ){','write-parameter }
')']
```

9.5 Goto statement

Description

Goto statements unconditionally transfer program execution to the statement prefixed by a label. Execution of a **goto statement** is terminated when it transfers program execution. If a **goto statement** transfers program execution to a statement outside of the current block then the call stack is unwound (i.e. functions and procedure calls are terminated) until the block containing the statement referred to by the **goto statement** is reached. So for example if the main program calls a procedure A, and procedure A calls procedure B, and procedure B calls procedure C, and a **goto statement** in procedure C transfers program execution back to a statement in procedure A then procedures C and B are terminated.

Standard Pascal (ISO/IEC 7185) describes the rules governing the use of **goto statement** as follows:

A label, if any, of a statement S shall be designated as prefixing S. The label shall be permitted to occur in a goto-statement G if and only if any of the following three conditions is satisfied.

```
a) S contains G.
b) S is a statement of a statement-sequence containing G.
c) S is a statement of the statement-sequence of the compound-statement
of the statement-part of a block containing G.
```

To fully understand these rules you will probably need a copy of the Standard (ISO/IEC 7185) but the following explanation should suffice for most people.

The first two rules cover **goto statements** that refer to statements in the current block, while the last rule covers **goto statements** that refer to statements in an enclosing block. The rules are basically saying that you can't use a **goto statement** to jump into the middle of a statement from outside of that statement. So for example the following code fragment is illegal.

```
goto 1;
for i := 1 to 10 do
1: writeln(i)
```

Example

Below is a simple program illustrating how to use goto statements.

```
program ten(output);
label loop;
var
count : integer;
begin
count := 1;
loop: writeln(count);
count := count + 1;
if count <= 10 then goto loop
end.
```

NOTE: The label **loop** used in this program is declared before it is used (which is a requirement for all labels).

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
goto-statement ='goto'label
```

9.6 Compound statement

Description

Compound statements group a statement-sequence (i.e. one or more statements) into a single statement, and are executed as follows:

1. Program execution is transfered to the first statement in the statement-sequence.
2. Program execution is transfered to the statements in the statement-sequence in the order in which they occur in the program text, unless diverted by one of the statements executed.
3. Execution of the **compound statement** is terminated when either the execution of the last statement in the statement-sequence is terminated, or when one of the statements in the statement-sequence transfers program execution to a statement outside of the **compound statement**.

You usually use **compound statements** when the rules of Pascal say that only one statement is allowed at a particular point in the program, but you want to put more than one statement there. The solution to this problem is to put the statements into a **compound statement**.

Example

For example look at the program fragment below:

```
write('Do you want to continue');
readln(answer);
if (answer = 'n') or (answer = 'N') then
begin
writeln('Program terminated by user');
halt
end
```

here we want to write a message to the screen and stop if the user answers **n** or **N**, but this requires two statements and the then-part of an if statement allows only a single statement. The solution as you can see is to put the two statements into a **compound statement**.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
compound-statement ='begin'statement-sequence 'end'

statement-sequence =statement { ';'statement }
```

9.7 If statement

Description

If statements either perform one of two actions, or conditionally perform a single action. **If statements** include the following:

- An if-condition, which is specified by a boolean expression (i.e. an expression of boolean type). The if-condition is evaluated, and its value determines which action, if any, is performed. If the value of the if-condition is true then the action specified by the statement in the then-part is performed. If the value of the if-condition is false and an else-part is present, then the action specified by the statement in the else-part is performed. If the value of the if-condition is false and an else-part is **not** present, then the execution of the **if statement** is terminated.
- A then-part, which contains a statement specifying an action to perform. This statement can be a compound statement, containing a statement-sequence. If this statement transfers program execution to a statement outside of the **if statement** then execution of the **if statement** is terminated when program execution is transfered.
- An optional else-part, which contains a statement specifying an action to perform. This statement can be a compound statement, containing a statement-sequence. If this statement transfers program execution to a statement outside of the **if statement** then execution of the **if statement** is terminated when program execution is transfered.

Example

For example

```
if x = y then
writeln('x is equal to y);
```

is an **if statement** that conditionally performs a single action (i.e. the action specified by the then-part). When this **if statement** is executed the if-condition (i.e. **x = y**) is evaluated and if true the statement in the then-part (i.e. **writeln('x is equal to y)**) is executed, and if false then execution of the **if statement** is terminated.

Here is another example

```
if x = y then
writeln('x is equal to y)
else
writeln('x is not to y);
```

is an **if statement** that performs one of two actions (i.e. either the action specified by the then-part or the action specified by the else-part). When this **if statement** is executed the if-condition (i.e. **x = y**) is evaluated, and if the value of the test condition is true then the statement in the then-part (i.e. **writeln('x is equal to y)**) is executed, and if the value of the test condition is false then the statement in the else-part (i.e. **writeln('x is not equal to y)**) is execuited.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
if-statement ='if'boolean-expression 'then'statement [else-part ]

else-part ='else'statement
```

9.8 Case statement

Description

Case statements perform one action out of a choice of actions, and include the following:

- A case-index, which is specified by an ordinal expression (i.e an expression of ordinal type). The case-index is evaluated and it's value determines which action is performed.
- Zero or more case-list-elements, each of which, represent a choice of action. Case-list-elements contain one or more case-constants and a statement. The case-constants in a case-list-element specify values which if equal to the value of the case-index will cause that case-list-element to be choosen. The statement in a case-list-element specifies the action performed when that case-list-element is choosen. This statement can be a compound statement, containing a statement-sequence. If this statement transfers program execution to a statement outside of the **case statement** then execution of the **case statement** is terminated when program execution is transfered.
- As an extension to Standard Pascal, Irie Pascal allows **case statements** to include case-statement-completers, which if present, represents a kind of default choice of action (i.e. the action choosen when none of the other actions are choosen). Case-statement-completers contain a statement-sequence, which specifies the action performed when the case-statement-completer is choosen. If one of the statements in the statement-sequence transfers program execution to a statement outside of the **case statement** then execution of the **case statement** is terminated when program execution is transfered.

Constant ranges can be used to specify a number of contiguous case-constants (for example 1..5 is the same as 1, 2, 3, 4, 5).

No two case-list-elements can contain the same constant, so

```
case x of
1, 2, 3 : write('1 to 3');
2 :  write('2');
end
```

is an error since both case-list-elements contain **2**.

Example

For example, the program fragment below shows an ordinal type and a procedure with a case statement.

```
type
DayOfWeek =
(monday, tuesday, wednesday, thursday, friday, saturday, sunday);

procedure WriteDayOfWeek(day : DayOfWeek);
begin
case day of
monday:    write('monday');
tuesday:   write('tuesday');
wednesday: write('wednesday');
thursday:  write('thursday');
friday:    write('friday');
saturday:  write('saturday');
sunday:    write('sunday');
end
end;
```

When the **case statement** is executed the case-index (i.e. **day**) is evaluated and if it is equal to **tuesday**, for example, then since the second case-list-element contains a constant equal to **tuesday** then the second case-list-element is choosen and it's statement (i.e. **write('tuesday')**) is executed.

Below is a slightly more complex example.

```
program example(input, output);
var
c : char;
begin
write('Enter a digit :');
readln(c);
case c of
'0'..'9' : writeln('OK');
'a'..'z', 'A'..'Z' : writeln('That is a letter');
otherwise writeln('What is that?');
end
end.
```

When the **case statement** is executed the case-index (i.e. **c**) is evaluated and its value used as follows:

- If the value is a digit then the first case-list-element is choosen, and it's statement (i.e. **writeln('OK')**) is executed.
- If the value is a letter then the second case-list-element is choosen, and it's statement (i.e. **writeln('That is a letter')**) is executed.
- If the value is not a digit or a letter then the case-statement-completer is choosen, and it's statement (i.e. **writeln('What is that?')**) is executed.

Irie Pascal implements case statements in two different ways depending on the type of the case-index.

1. If the case-index in the **case statement** has a type with 1024 values or less then Irie Pascal implements the **case statement** using a jump table.
2. If the case-index in the **case statement** has a type with more than 1024 values then Irie Pascal implements the **case statement** as a series of hidden if statements.

In earlier versions of Irie Pascal the second (if statement) implementation had serious problems, the rule that no two case-list-elements can have the same constant was **not** enforced, and the case-index was evaluated once for each case-list-element until a match was found. As a result of these problems a warning message was issued when the second implementation was used. These problems have now been removed from the second implementation, so that now no matter what implementation is used, the rule that no two case-list-elements can have the same constant is **always** enforced, and the case-index is **never** evaluated more than once. As a result the warning message is no longer issued when the second implementation is used.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
case-statement ='case'case-index case-body [';']'end'

case-index =ordinal-expression

case-body = case-list-elements [[';']case-statement-completer ]|
case-statement-completer

case-list-elements =case-list-element {';'case-list-element }

case-list-element =case-constant-list ':'statement

case-constant-list =case-specifier {','case-specifier }

case-specifier =case-constant ['..'case-constant ]

case-constant =ordinal-constant

case-statement-completer =('otherwise'|'else')statement-sequence
```

9.9 Repeat statement

Description

Repeat statements perform an action in a conditional loop until the loop condition is true, and include the following:

- A statement-sequence that specifies the action performed in the loop. If a statement in the statement-sequence transfers program execution to a statement outside of the **repeat statement** then execution of the **repeat statement** is terminated when program execution is transfered.
- The loop condition, which is specified by an expression of boolean type. After each iteration of the loop the loop condition is evaluated and if it's value is false the loop continues, but if it's value is true the loop terminates.

NOTE: The loop condition is evaluated after the statement-sequence is executed so the statement-sequence is executed at least once. Usually the statement-sequence will perform some action that will eventually cause the loop condition to be true and terminate the loop.

Example

For example, below is a very simple program which illustrates the use of the "repeat" statement.

```
program ten(output);
```

```
var
count : 1..11;
begin
count := 1;
repeat
writeln(count);
count := count + 1
until count > 10
end.
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
repeat-statement ='repeat'statement-sequence 'until'boolean-expression

statement-sequence =statement {';'statement }
```

9.10 While statement

Description

While statements perform an action in a conditional loop as long as the loop condition is true, and include the following:

- The loop condition, which is specified by an expression of boolean type. Before each iteration of the loop the loop condition is evaluated and if it's value is true the loop continues, but if it's value is false the loop terminates.
- A statement that specifies the action performed in the loop. This statement can be a compound statement, containing a statement-sequence. If this statement transfers program execution to a statement outside of the **while statement** then execution of the **while statement** is terminated when program execution is transfered.

NOTE: The loop condition is evaluated before the statement in the **while statement** is executed so this statement may never be executed. Usually the statement in the **while statement** will perform some action that will eventually cause the loop condition to be false and terminate the execution of the loop.

Example

For example below is a very simple program which illustrates the use of the "while" statement.

```
program ten(output);
var
count : 1..11;
begin
count := 1;
while count <= 10 do
begin
writeln(count);
count := count + 1;
end
end.
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
while-statement ='while'boolean-expression 'do'statement
```

9.11 For statement

Description

For statements perform an action in a *counting loop* (i.e. perform the action for a specified number of times and counts each performance of the action). **For statements** contain the following:

- A variable (called the control variable) that stores the current loop counter.
- The initial value of the loop counter, which is specified by an expression of ordinal type.
- The direction of the loop count, specified either by the reserved word **to** which indicates that the loop counts upwards, or by the reserved word **downto** which indicates that the loop counts downwards.
- The final value of the loop counter, which is specified by an expression of ordinal type.
- A statement that specifies the action performed in the *counting loop*. This statement can be a compound statement, containing a statement-sequence.

Usually execution of a **for statement** is as follows:

1. The initial value is stored in the control variable.
2. The statement contained in the **for statement** is executed.
3. If the value stored in the control variable is equal to the final value then execution of the **for statement** is terminated.
4. The value stored in the control variable is incremented by one (in loops counting upwards), or decremented by one (in loops counting downwards).
5. Execution of the **for statement** continues with step 2.

However the following situations can affect the execution of a **for statement**.

- If the statement contained in the **for statement** transfers program execution to a statement outside of the **for statement** then execution of the **for statement** is terminated when program execution is transfered.
- The **for statement's** *counting loop* can specify that the action be performed zero times, in which case no value is stored in the control variable and the statement contained in the **for statement** is not executed. This situation occurs when either the initial value is greater than the final value in loops counting upwards, or the initial value is less than the final value in loops counting downwards (in either case the initial value and the final value are evaluated and compared but no other action is performed by the **for statement** before it's execution is terminated.

You should not depend on the control variable having any particular value after the execution of a **for statement** is terminated, unless program execution was transfered out of the **for statement** by a goto statement.

Normally when executing a **for statement** it is best if the value of the control variable is changed only by the **for statement**. If another statement alters the value of a **for statement's** control variable it usually causes problems in the program. In an apparent effort to protect programmers from making that kind of mistake Standard Pascal (ISO/IEC 7185) defines a number of rules that restrict how control variables are used, and Irie Pascal implements these rules.

The first rule is that control variables must be declared at the same level as the **for statement** containing them (i.e. if the **for statement** is in a function or procedure then the control variable must be local to that function or procedure, and if the **for statement** is in the main program block then the control variable must be global). Keeping the control variable local makes it easier to control access.

The second rule is that neither the **for statement** nor any function or procedure local to the block containing the **for statement** shall contain a statement that *threatens* the control variable. A statement *threatens* the control variable if the execution of the statement could possibly alter the value stored in the control variable.

- An assignment statement *threatens* a control variable if the control variable is the variable being assigned to.
- A function or procedure call *threatens* a control variable if the control variable is being passed by reference.
- A read or readln statement *threatens* a control variable if the control variable is passed as an actual parameter to these procedures.
- A **for statement** *threatens* it's own control variable.

Example

The following program uses a **for statement** to count from 1 to 10.

```
program CountToTen(output);

procedure CountToTen;
var
i : 1..10;
begin (* CountToTen *)
for i := 1 to 10 do
writeln(i);
end; (* CountToTen *)

begin
CountToTen;
end.
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
for-statement ='for'control-variable ':='initial-value ('to'|'downto')final-value
'do'statement

control-variable =entire-variable

initial-value =ordinal-expression

final-value =ordinal-expression
```

9.12 With statement

Description

With statements perform an action, while allowing the fields of specified record variables to be referencing using their field identifiers only. **With statements** include the following:

- A record-variable-list, which contains one or more record variables.
- A statement which specifies the action to perform. Inside this statement the fields, in any of the record variables in the record-variable-list, can be referenced using their field identifiers only.

Example

For example give the following type and variable.

```
type
student = record
name : string;
address : string;
grade : integer;
end;

var
s : student;
```

then the following **with statement**

```
with s do
begin
name := 'John';
address := 'main street';
grade := 20;
end
```

is equivalent to

```
begin
s.name := 'John';
s.address := 'main street';
s.grade := 20;
end
```

Since some record types can contain other record types you can have nested **with statements** like the following:

```
with record1 do
with record2 do
with record3 do
statement
```

or you can use the following shorthand which does the same thing

```
with record1, record2, record3 do
statement
```

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
with-statement ='with'record-variable-list 'do'statement

record-variable-list =record-variable { ';'record-variable }
```

9.13 Procedure method statement

Description

Procedure method statements are similiar to procedure statements, the only difference is that the procedures executed by **procedure method statements** are methods of instance variables (see object variables for more information about instance variables) instead of normal procedures.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
procedure-method-statement =procedure-method-specifier [actual-parameter-list ]

actual-parameter =expression |variable-access |
procedure-identifier |function-identifier

actual-parameter-list ='('actual-parameter {','actual-parameter }')'

procedure-method-identifier =identifier

procedure-method-specifier =object-variable '.'procedure-method-identifier
```

10.1 Compatible types

Irie Pascal implements the type compatibility rules defined by Standard Pascal (ISO/IEC 7185), and adds one more rule to support variable length string types. The first four rules below are taken from ISO/IEC 7185 and the last rule is added to support variable length string types.

Types T1 and T2 shall be designated compatible if any of the following statements is true:

1. T1 and T2 are the same type.
2. T1 is a subrange of T2, or T2 is a subrange of T1, or both T1 and T2 are subranges of the same host type.
3. T1 and T2 are set-types of compatible base-types, and either both T1 and T2 are designated packed or neither T1 nor T2 is designated packed.
4. T1 and T2 are fixed-length-string-types with the same number of components.
5. Either T1 is a variable-length-string-type and T2 is a variable-length-string-type, or T1 is a variable-length-string-type and T2 is a fixed-length-string-type, and the number of components in T2 is less than or equal to the maximum length of T1, or T1 is a variable-length-string-type and T2 is the char-type or T2 is a variable-length-string-type and T1 is a fixed-length-string-type, and the number of components in T1 is less than or equal to the maximum length of T2, or T2 is a variable-length-string-type and T1 is the char-type.

10.2.1 What is assignment compatibility?

Pascal is a strongly-typed language which means, amoung other things, that every value used in a Pascal program has a type, and each value's type controls where and how it can be used. One way Pascal controls where and how values can be used is through the concept of **assignment compatibility**. Assignment compatibility is a relationship between values and types, and for any given value and type the relationship either exists between them or is does not (i.e. the value is either assignment compatible with the type or it is not).

Irie Pascal implements the assignment compatibility rules defined by Standard Pascal (ISO/IEC 7185), and adds one more rule to support variable length string types. The first five rules below are taken from ISO/IEC 7185 and the sixth one was added to support variable length string types.

A value (V) of type T2 shall be designated *assignment compatible* with a type T1 if any of the following six statements is true:

1. T1 and T2 are the same type, and that type is not a file type for a type that contains a file type.
2. T1 is the real type and T2 is an integral type.
3. T1 and T2 are compatible ordinal types, and V is in the closed interval specified by the type T1.
4. T1 and T2 are compatible set types, and all the members of V are in the closed interval specified by the base-type of T1.
5. T1 and T2 are fixed length string types with the same length.
6. T1 is a variable length string type and either T2 is a variable length string type, or T2 is a fixed length string type, or T2 is the char type. In addition the length of V must not be greater then the maximum length of T1.

Assignment compatibility is used to control where and how values can be used, in the following situations:

- Assignment compatibility with array indexing
- Assignnment compatibility with value parameters
- Assignment compatibility with "read"
- Assignment compatibility with assignment statements
- Assignment compatibility with "for" statements
- Assignment compatibility with transfer procedures

10.2.2 Assignment compatibility with array indexing

When an element of an array variable is being referenced, the value of the index expression shall be assignment compatible with the index type of the array variable.

For example given:

```
var x : array[1..10] of integer;
```

and

```
x[i] := 20;
```

then **i** (the value of the index expression) shall be assignment compatible with **1..10** (the index type of the array variable).

10.2.3 Assignnment compatibility with value parameters

When passing an actual parameter by value to a formal parameter of a function or procedure, the actual parameter shall be assignment compatible with the type of the formal parameter.

For example given:

```
function IsDigit(c : char) : Boolean;
begin
if c in ['0'..'9'] then
IsDigit := true
```

```
else
IsDigit := false
end;
```

and

```
while not IsDigit(key) do
read(key)
```

then **key** (the actual parameter in **IsDigit(key)** shall be assignment compatible with **char** (the type of the formal parameter).

10.2.4 Assignment compatibility with "read"

When using the built-in procedure read to retrieve a value from a file into a variable, the value shall be assignment compatible with the type of the variable.

For example given:

```
var s : string[40];
```

and

```
read(s);
```

then the value read in by

`read(s)`

must be assignment compatible with

`string[40]`

.

10.2.5 Assignment compatibility with assignment statements

The value of the expression on the right-hand side of an assignment statement shall be assignment compatible with the type of the left-hand side.

For example given:

```
var age : 0..100;
```

and

```
age := v;
```

then **v** (the right-hand side of the assignment statement) shall be assignment compatible with **0..100** (the type of the left-hand side of the assignment statement).

10.2.6 Assignment compatibility with "for"

The initial value and the final value in a for statement shall be assignment compatible with the type of the control variable, if the statement contained in the for statement is executed.

For example given:

```
for i := low to high do writeln(i);
```

then

`low``(the initial value)`

and

`high``(the final value)`

shall be assignment compatible with the type of

`i``(the control variable)`

if

`writeln(i)``(the statement in the for statement)`

is executed. **NOTE:** **writeln(i)** will get executed unless **low** is greater than **high**.

10.2.7 Assignment compatibility with transfer procedures

In the statements

`pack(a,i,z)`

and

`unpack(z,a,i)`

the value of **i** shall be assignment compatible with the index type of the array variable **a**.

For example given:

```
var a : array[1..100] of real;
z : packed array[1..100] of real;
i : integer;
```

and

```
pack(a,i,z) or unpack(z,a,i)
```

then **i** shall be assignment compatible with

`1..100``(the index-type of `**`a`**`).`

11.1 What are program parameters?

Description

Program parameters are identifiers which appear at the top of the program, in the program heading, after the program name and between parentheses. In addition to appearing after the program name, most program parameters must also appear in a variable declaration in the main program block. Two special program parameters input and output are the exceptions to this rule. The appearance of the two special program parameters in the program heading automatically declares them as file variables, associated with the standard-input and standard-output streams.

You can use program parameters to access the command-line arguments passed to your program.
NOTE: You can also use paramcount and paramstr to access command-line arguments.

Example

For example, suppose you want to write a program to append two files together, writing the appended files to a third file, then you might write a program similar to the program below.

```
(*********************************************************************
** This program appends two files together, writing the appended files
** out to a third file.
*)
program append(in1, in2, out, output);
type
char_file = file of char;
var
in1 : char_files;      (* first input file *)
in2 : char_files;      (* second input file *)
out : char_files;      (* output file *)

(**************************************************************
** PURPOSE: Writes copies the contents of one file into another.
** ARGUMENTS:
**    'f' - the input file
**    'g' - the output file
** NOTES: It is up to the caller to open and close the files.
*)
procedure WriteFile(var f, g: char_file);
var
c : char;
begin
while not eof(f) do
begin
read(f, c);
write(g, c)
end
end;

(**********************************************
** PURPOSE: Writes a help screen and then halts
*)
procedure syntax;
begin
writeln('Appends two files together and writes the output to a third file');
writeln('Syntax');
writeln('   ivm append in1 in2 out');
writeln('where "in1" is the first input file');
writeln('and   "in2" is the second input file');
writeln('and   "out" is the output file');
```

```
halt
end;

begin
if paramcount <> 3 then
syntax;
rewrite(out);
reset(in1);
WriteFile(in1, out);
close(in1);
reset(in2);
WriteFile(in2, out);
close(in2)
close(out);
end.
```

The first thing to notice about this program is the line below:

```
program append(in1, in2, out, output);
```

Here four program parameters are being used:

1. **in1** - accesses the first command-line argument.
2. **in2** - accesses the second command-line argument.
3. **out** - accesses the third command-line argument.
4. output - does not access a command-line argument.

Since output does not access a command-line argument it can appear in any position (not just the last position), without changing which command-line arguments are accessed by the other program parameters. In other words all of the following are equivalent

```
program append(output, in1, in2, out);
```

and

```
program append(in1, output, in2, out);
```

and

```
program append(in1, in2, output, out);
```

and

```
program append(in1, in2, out, output);
```

The second thing to notice about the program are the following lines.

```
var
in1 : char_files;      (* first input file *)
in2 : char_files;      (* second input file *)
out : char_files;      (* output file *)
```

Here the program parameters (except for output) appear in variable declarations, in the main program block, as required. They are declared to be variables of type **char_file** which has been declared to be

```
file of char
```

Now since they are declared to be file variables it means that the command-line arguments accessed specify the initial names of the file variables.

The final thing to notice about the program are the following lines.

```
if paramcount <> 3 then
syntax;
```

These lines cause the procedure **syntax** to be called if the number of command-line arguments is not 3. These lines are intended to prevent problems if the user does not enter 3 command-line arguments.

So far only file type program parameters have been described. You can also use string type program parameters. They can also access command-line arguments, but in a different way. The command-line argument accessed is simply assigned to the string type program parameter.

If you use program parameters other than file type or string type then the command-line argument is ignored. The compiler will issue a warning, that the command-line argument has an invalid type, but otherwise do nothing.

For example look at this rather strange program.

```
program strange(f, s1, dummy, s2);
var
f : text;
s1, s2 : string;
dummy : real;
begin
rewrite(f);
writeln(f, s1);
writeln(f, s2)
end.
```

If you compile it, you will get some warnings but ignore them. If you run the program with

```
ivm strange out.txt first skip second
```

then the first program parameter **f** will access **out.txt**, and since **f** is a file type program argument, when rewrite is used on **f**, the file **out.txt** will be opened. The second program parameter **s1** will access **first**, and since this is a string type program argument, then **first** will be assigned to **s1**. The third program parameter **dummy** is not a file type or string type program parameter so the third command-line argument **skip** will be ignored. The fourth program parameter **s2** will access **second**, and since this is a string type program argument then **second** will be assigned to **s2**.

So the effect of the following three lines

```
rewrite(f);
writeln(f, s1);
writeln(f, s2)
```

is that a text file **out.txt** is opened and two lines are written to it. The first line will be **first** and the second will be **second**.

Syntax

(**NOTE:** for clarity some parts of the syntax are omitted, see Irie Pascal Grammar for the full syntax):

```
program =program-heading ';'program-block

identifier =letter {letter |digit }

identifier-list =identifier {','identifier }
```

```
program-block =block

program-heading ='program'identifier ['('program-parameter-list ')']

program-parameter-list =identifier-list
```

12.1 What are sockets?

***NOTE:** The information provided in this manual about sockets is just an introduction, and is intended to give you a basic understanding of how to write Irie Pascal programs that use sockets. If you intend to write serious programs (i.e. programs that you use for important purposes and which must be robust and reliable) that use sockets, then you should consult more advanced and complete sources of information about sockets.*

Introduction To Sockets

Sockets were first introduced by the University of California, Berkeley, in their version of UNIX, as a method for processes (i.e. running programs) to communicate with each other. Sockets are now widely supported by many operating systems, and the concept has been expanded to include processes communicating over a network.

Sockets provide a useful abstraction that insulates communicating processes, to some extent, from the details about how the communication is done. Programs can basically treat each socket as an end-point in a communications link.

Most programs that use sockets follow a client/server model, and are either clients or servers. The major difference between client and server socket programs is the way in which they establish connections with each other. Basically a server socket program creates a socket and then uses this socket to listen for attempts by client socket programs to connect. A client socket program, on the other-hand creates a socket and then tries to connect this socket to the socket created by the server program. Once a connection has been established, the difference between client and server socket programs becomes less important because data can be sent and received between sockets, in both directions.

Socket Names

The term *name*, when applied to sockets, means the unique reference used to identify sockets within an address family. The elements that make up the *name* of a socket could vary depending on the address family the socket belongs to. Most of the sockets that you are likely to use will belong to the internet address family, and their *names* will be made of the following three elements:

1. Address family - Identifes the name as belonging to the internet address family
2. Host address - IP address (e.g. 127.0.0.1)
3. Port number - TCP/IP port number

Socket Types

Stream Sockets

Stream sockets are connection-oriented, and must be in a connected state (i.e. they must have established a connection to another socket) before any data can be sent or received. A connection to another socket is created with a call to the connect function. Once connected, data can be transferred, between the sockets, using the send, sendto, recv, and recvfrom functions.

Datagram Sockets

Datagram sockets are not connection-oriented, and are used to send and receive blocks of data, called *datagrams*.

In order to send *datagrams*, both the source socket and the *name* of one or more destination sockets, must be specified. The sendto function, which allows you to specify both a source socket and the *name* of one or more destination sockets, is normally used to send *datagrams*. The send function can also be used to send *datagrams*, however because the send function does not allow you to specify the *name* for the destination datagram socket or sockets, the *name* of the destination datagram socket or sockets must have been previously specified by the connect function.

The recvfrom function can be used to receive *datagrams*, and can optionally return the *name* of the socket that sent the *datagram*. The recv function can also be used to receive *datagrams*, however the connect function must have been used previously to specify the *name* for the source datagram socket or sockets.

Reference Information

As you might expect, the internet is an excellent source of information about sockets. The following two web sites seemed to contain particularly useful information about sockets, at the time this manual was created:

- www-users.cs.umn.edu/~bentlema/unix/advipc/ipc.html
- www.sockaddr.com/TheSocketsParadigm.html

12.2 Introducing The WinSock2 library

The WinSock2 Library

Support for sockets has been included with the Windows operating system by an interface library called *WinSock* or more recently *WinSock2*. The older *WinSock* library will not be discussed any further in this manual. The WinSock2 library supports most of the functionality provided by Berkeley Sockets (usually using the same names for functions, types, and constants). The WinSock2 library also supports many functions not present in the Berkeley Sockets library.

The WinSock2 library is implemented as a Windows DLL and is distributed along with a C header file (which provides declarations for the WinSock2 functions, types, and constants). You can use Irie Pascal to write programs that use the WinSock2 library. The WinSock2 C header file has been translated into Pascal, and is distributed with Irie Pascal for your convenience. You should include the translated file (i.e. **WinSock2.inc**) in your programs that use the WinSock2 library. A number of sample programs, that use sockets, are included with Irie Pascal in the ***InstallDir*\samples\sockets** folder. **NOTE:** ***InstallDir*** is the folder where Irie Pascal was installed.

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.3 Initializing the WinSock2 library

The WSAStartUp Function

Description

The **WSAStartUp** function initializes the WinSock2 library for use by your program, and must be the first WinSock2 library function called by your program. Calling the **WSAStartUp** function serves two major purposes:

- The first purpose is to load the DLL that implements the WinSock2 library (i.e. **Ws2_32.dll**). **NOTE:** The other WinSock2 library functions expect the WinSock2 library DLL to be loaded before they are called.
- The second purpose for calling the **WSAStartUp** function is to establish what version of the Windows Sockets API your program should use.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **WSAStartUp** function:

```
function WSAStartup(wVersionRequested : shortword; var lpWSAData : WSADATA) :
integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **WSAStartUp** function is an expression of shortword type. This argument indicates the highest version of the Windows Sockets API that your program can support. Bits 0 through 7 (i.e. the least significant byte) specify the major version number of the API. Bits 8 through 15 (i.e. the most significant byte) specify the minor version number of the API. So to specify that the highest Windows Socket API version, that your program can support, is 2.1, you could use **2+1*256**, or **$0102**, or just **258** as the value of this argument.

The Second Argument

The second argument passed to the **WSAStartUp** function, is a reference to a variable of type *WSADATA*. This argument is used by the **WSAStartUp** function to pass information back to your program. **NOTE:** The *WSADATA* type is declared in the system include file **WinSock2.inc**, as a record type, with the following important fields:

1. **wVersion**, contains the version of the Windows Sockets API that your program should use (the least significant byte contains the major version number, and the most significant byte contains the minor version number).
2. **wHighVersion**, contains the highest version of the Windows Sockets API that can be supported by the WinSock2 library DLL (the layout of this field is the same as the layout of the first field).
3. **szDescription**, contains a description of the Windows Sockets API implementation.
4. **szSystemStatus**, contains status or configuration information that might be useful to the user or

the support staff (this field may be empty).

Return Values

The **WSAStartUp** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A non-zero value is an error code that indicates the reason why the call failed.

Sample Program

The sample program below illustrates how to call the **WSAStartUp** and **WSACleanUp** functions.

```
(*$I winsock2.inc *)
program start(output);
var
data : WSADATA;
iRet : integer;
begin
iRet := WSAStartUp($0202, data);
if iRet <> 0 then
begin
writeln('WSAStartUp call failed. Return Code=', iRet);
halt;
end;
writeln('WSAStartUp call succeeded.');
writeln('Version: $', hex(data.wVersion));
writeln('High Version: $', hex(data.wHighVersion));
writeln('Description: ', data.szDescription);
writeln('System Status: ', data.szSystemStatus);
iRet := WSACleanUp
end.
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.4 Cleaning up WinSock2

The WSACleanUp Function

Description

The **WSACleanUp** function is used to indicate that your program has finished using the WinSock2 library, and should be the last WinSock2 library function your program calls. Calling the **WSACleanUp** function allows the WinSock2 library to free any resources allocated on behalf of your program. Any sockets that are open when the **WSACleanup** function is called are reset and automatically deallocated as if the closesocket function was called. Calling the **WSACleanup** function may result in loss of data, if there are sockets that have been closed with the closesocket function but still have pending data to be sent, and the call to the **WSACleanup** function causes the WinSock2 library DLL to be unloaded from memory. In order to prevent loss of data, your program should call the shutdown function before calling the closesocket function and the **WSACleanUp** function.

There must be a call to the **WSACleanUp** function for every successful call to the WSAStartup function made by your program. Only the final call to the **WSACleanUp** function does the actual cleanup. The preceding calls simply decrement an internal reference count in the **Ws2_32.dll**.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **WSACleanUp** function:

```
function WSACleanup : integer;
external dll='ws2_32.dll';
```

Arguments

None.

Return Values

The **WSACleanUp** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail.

Sample Program

The sample program below illustrates how to call the WSAStartup and **WSACleanUp** functions.

```
(*$I winsock2.inc *)
program start(output);
var
data : WSADATA;
iRet : integer;
begin
iRet := WSAStartUp($0202, data);
if iRet <> 0 then
begin
writeln('WSAStartUp call failed. Return Code=', iRet);
halt;
end;
writeln('WSAStartUp call succeeded.');
writeln('Version: $', hex(data.wVersion));
writeln('High Version: $', hex(data.wHighVersion));
writeln('Description: ', data.szDescription);
writeln('System Status: ', data.szSystemStatus);
iRet := WSACleanUp
end.
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.5 Checking for errors

The WSAGetLastError Function

Description

The **WSAGetLastError** function retrieves the error code for the last unsuccessful call to a WinSock2 library function. Most WinSock2 library functions indicate failure by returning a particular value, and you have to call the **WSAGetLastError** function to find out the reason for the failure. The error code returned by this function is not effected by calls to WinSock2 library functions that succeed, or by calling this function to retrieve the error code. An error code of zero means that no WinSock2 library function has failed since you initialized the WinSock2 library (i.e. called the WSAStartUp function), or since you last set the error code to zero by calling the **WSASetLastError** function.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **WSAGetLastError** function:

```
function WSAGetLastError : integer;
external dll='ws2_32.dll';
```

Arguments

None.

Return Values

See Description.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.6.1 Creating data sockets

Data Sockets

In this manual, the term *data sockets* means sockets that you can use to send and receive data. The other kind of sockets are *listening sockets*, which you can use to listen for attempts by other sockets to connect. When you first create a socket it is a *data socket* by default. In order to create a *listening socket* you must take some extra steps (see creating listening sockets).

The CreateSocket Function

Description

The name of the function, in the Berkeley and Windows Sockets libraries, that creates sockets is **socket**, and the name of the return type of this function is **SOCKET**. Although the two names differ only in case, this is not a problem for these libraries because the libraries are designed for the C programming language, which is case sensitive. In C, **socket** and **SOCKET** are different identifiers, and so there is no conflict between the name of the function and the name of its return type. However, Pascal is not a case sensitive language, so **socket** and **SOCKET** are the same identifier and the name of the function would conflict with the name of the return type. In order to avoid this conflict the name of the function was changed to **CreateSocket**.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **CreateSocket** function:

```
function createsocket(af, typ, protocol : integer) : SOCKET;
external dll='ws2_32.dll' name='socket';
```

Arguments

The First Argument

The first argument passed to the **CreateSocket** function is an expression of integer type. This argument indicates the address family the socket should belong to. Normally you would use the constant *AF_INET* as the value of the first argument. **NOTE:** *AF_NET* is declared in the system include file **WinSock2.inc**.

The Second Argument

The second argument passed to the **CreateSocket** function is an expression of integer type. This argument indicates the kind of socket to create and is normally one of the following constants.

- *SOCK_STREAM* - connection orientated stream socket
- *SOCK_DGRAM* - connectionless datagram socket
- *SOCK_RAW* - raw socket

NOTE: These constants are declared in the system include file **WinSock2.inc**.

The Third Argument

The third argument passed to the **CreateSocket** function is an expression of integer type. This argument indicates the protocol to use for the socket from the address family. Normally this argument is zero for all kinds of sockets except raw sockets.

Return Values

The **CreateSocket** function returns the created socket if the call is successful. If the call fails then the value *INVALID_SOCKET* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** *INVALID_SOCKET* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.6.2 Creating listening sockets

Listening Sockets

Listening sockets are sockets that you can use to *listen* for attempts by other sockets to establish a connection. Usually listening sockets are created by server socket programs that need to know when client socket programs want to connect. In order to create listening socket, you need to do three things:

1. Create a data socket (see creating a data socket)
2. Bind the socket to a local address (see the bind function)
3. Convert the data socket into a listening socket (see the listen function)

12.7.1 The bind function

Description

The **bind** function assigns a local *name* to a socket.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **bind** function:

```
function bind(s : SOCKET; name : address; namelen : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **bind** function is the *unnamed* socket to which you want to assign a *name*.

The Second Argument

The second argument passed to the **bind** function is the *name* to be assigned to the socket. The type of this argument is address which means that you must use the built-in function addr to pass the operating system address of the *name*. This type is choosen because the format of a socket's *name* could depend on what address family the socket belongs to, and using the address type leaves it up to you to choose the appropriate type. Usually the socket will belong to the internet address family and you can use a variable of type *sockaddr_in* to specify the *name*. *sockaddr_in* is a record type with the following four fields:

- *sin_family* - The address family (usually *AF_INET*)
- *sin_addr* - The IP address
- *sin_port* - The port number
- *sin_zero* - Eight bytes of zeroes

If you do not care what IP address is assigned to the socket then you should specify the constant *INADDR_ANY* for the *sin_addr* field. This allows the underlying network service provider to choose an appropriate IP address. If you do not care what port number is used then you should specify zero as the *sin_port* field. This allows the underlying network service provider to choose an appropriate port number. **NOTE:** The constant *INADDR_ANY* and the type *sockaddr_in* are declared in the system include file **WinSock2.inc**.

The Third Argument

The third argument passed to the **bind** function is the length (in bytes) of the *name* to be assigned to the socket.

Return Values

The **bind** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Example

The following example function is taken from the sample sockets programs distributed with Irie Pascal, and demonstrates how to create a socket, bind it, and convert it into a listening socket.

```
//This function creates a listening socket.
//First it creates a socket. Then it binds the socket to
//a particular port (i.e. the port we want to listen on).
//The socket is not bound to any particular address (INADDR_ANY is used instead)
//because connections will be accepted from any IP address.
//Finally "listen" is called to convert the socket to a listing socket.
function CreateListeningSocket(port : integer) : socket;
var
s : socket;
sa : sockaddr_in;
iRet : integer;
begin (* CreateListeningSocket *)
s := createsocket(AF_INET, SOCK_STREAM, 0);
if s = INVALID_SOCKET then
FatalSocketError('CreateSocket call failed');
```

```
fill(sa, 0);
sa.sin_family := AF_INET;
sa.sin_port := htonl(port) shr 16;
sa.sin_addr := htonl(INADDR_ANY) shr 16;
iRet := bind(s, addr(sa), sizeof(sa));
if iRet <> 0 then
FatalSocketError('Call to bind failed');
iRet := listen(s, SOMAXCONN);
if iRet <> 0 then
FatalSocketError('Call to listen failed');
CreateListeningSocket := s;
end; (* CreateListeningSocket *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.8.1 The listen function

Description

The **listen** function converts a *data socket* into a *listening socket*. In this manual, the term *data socket* means a socket that you can use to send and receive data, and the term *listening socket* means a socket that you can use to listen for attempts by other sockets to connect. When you first create a socket it is a *data socket* by default.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **listen** function:

```
function listen(s : SOCKET; backlog : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **listen** function is the *data socket* socket that you want to convert into a *listening socket*.

The Second Argument

The second argument passed to the **listen** function is the maximum length of the queue of pending connections. If you pass a value, for this argument, that is too large then the underlying service provider will choose an appropriate value. If you are not sure what value you should use for this argument then use *SOMAXCONN*, which will cause the underlying service provider to choose an appropriate value.
NOTE: The constant *SOMAXCONN* is declared in the system include file **WinSock2.inc**.

Return Values

The **listen** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Example

The following example function is taken from the sample sockets programs distributed with Irie Pascal, and demonstrates how to create a socket, bind it, and convert it into a listening socket.

```
//This function creates a listening socket.
//First it creates a socket. Then it binds the socket to
//a particular port (i.e. the port we want to listen on).
//The socket is not bound to any particular address (INADDR_ANY is used instead)
//because connections will be accepted from any IP address.
//Finally "listen" is called to convert the socket to a listing socket.
function CreateListeningSocket(port : integer) : socket;
var
s : socket;
sa : sockaddr_in;
iRet : integer;
begin (* CreateListeningSocket *)
s := createsocket(AF_INET, SOCK_STREAM, 0);
if s = INVALID_SOCKET then
FatalSocketError('CreateSocket call failed');
fill(sa, 0);
sa.sin_family := AF_INET;
sa.sin_port := htonl(port) shr 16;
sa.sin_addr := htonl(INADDR_ANY) shr 16;
iRet := bind(s, addr(sa), sizeof(sa));
if iRet <> 0 then
FatalSocketError('Call to bind failed');
iRet := listen(s, SOMAXCONN);
if iRet <> 0 then
FatalSocketError('Call to listen failed');
CreateListeningSocket := s;
end; (* CreateListeningSocket *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.9.1 Dealing with blocking

Blocked Function Calls

Some WinSock2 library functions can become blocked waiting for one or more required conditions to be satisfied. For example, the recv function, which reads data from sockets, requires that there be data in the sockets to be read. The recv function will become blocked, if it is called to read data from a socket, and the socket's non-blocking mode is disabled (see non-blocking mode below), and there is no data in the socket. In this case the recv function will not return to the caller until there is data in the socket, or until the connection to the socket is closed at the other end. **NOTE:** If your program calls a WinSock2 library function that becomes blocked then your program will also be blocked waiting for the function to become unblocked and return from the call.

Sometimes you might want to allow your program to become blocked. For example if the sole purpose of your program is to read some data and process it, and your program can't sensibly continue until it has read the data, then you might choose to allow your program to become blocked if no data is available to be read. **NOTE:** When your program is blocked it does not use up any processor cycles, and so in the example above, being blocked can be a very effecient way for your program to wait until there is data for it to process.

On the other hand, sometimes you will not want your program to ever become blocked. This is true if your program must be always responsive. For example, if your program is a service, then it must always be ready to respond to a request to stop.

Non-Blocking Mode

Sockets have a non-blocking mode that can be enabled or disabled. When a socket is created, it's non-blocking mode is disabled. One way to prevent WinSock2 library functions from becoming blocked is to use sockets with non-blocking mode enabled. If a WinSock2 library function is called, on a socket with non-blocking mode enabled, and conditions are such that the function would block were non-blocking mode disabled, then instead of becoming blocked the function call will fail. If you call WSAGetLastError after the function call fails, the return code will be *WSAEWOULDBLOCK*. So for example, if you call the recv function to read data from a socket, and the socket has non-blocking mode enabled, and there is no data waiting to be read from the socket, then the recv function will return *SOCKET_ERROR* to indicate that the call failed. If you then call the WSAGetLastError function, the error code returned will be *WSAEWOULDBLOCK*. **NOTE:** The constant *WSAEWOULDBLOCK* is declared in the system include file **WinSock2.inc**.

How To Avoid Getting Blocked

There are basically two ways you can ensure that your program will never becoming blocked:

1. You can enable non-blocking mode for sockets that might otherwise become blocked. See the ioctlsocket function for more information.
2. You can make sure that the conditions that might cause a socket to block never occur. The conditions that might cause a socket to block are:
   - Trying to read data from a socket when there is no data to read.
3. Trying to write data to a socket when the trasport system has no space to hold the data.
4. Trying to accepting a connection when there are no connection requests waiting to be accepted.
   See the select function for more information.

12.9.2 The ioctlsocket function

Description

The **ioctlsocket** function sends commands to sockets. See the description of the function arguments below, for information about the commands that can be sent.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **ioctlsocket** function:

```
function ioctlsocket(s : SOCKET; cmd : integer; var argp : u_long) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **ioctlsocket** function is the socket that you want to send a command to.

The Second Argument

The second argument passed to the **ioctlsocket** function is the command that you want to send to the socket. The Windows Sockets API supports three commands, and these commands are:

1. FIONBIO - Enables or disables the socket's non-blocking mode. The value of the third argument specifies whether non-blocking mode is enabled or disabled. Use a non-zero value for the third argument to enable non-blocking mode, and zero to disable non-blocking mode. **NOTE:** When a socket is created, it's non-blocking mode is disabled.
2. FIONREAD - Returns the amount of data that can be read from the socket in a single call. This is not necessarily the same as the total amount of data waiting to be read from the socket. For datagram sockets the value returned by this command is the size of the first datagram waiting to be read. **NOTE:** The value returned by this command is placed in the third argument to this function.
3. SIOCATMARK - Returns whether or not all Out Of Band (OOB) data has been read. This command returns a non-zero value if there is no OOB data waiting to be read, and zero if there is OOB data waiting to be read. A full description of OOB data is outside the scope of this manual, however OOB data is basically data that is sent with high priority. OOB data can *jump the queue* and be read at the other end before normal data that was sent previously. **NOTE:** OOB data and normal data are never read together in the same call. **NOTE:** The value returned by this command is placed in the third argument to this function.

NOTE: The constants FIONBIO, FIONREAD, and SIOCATMARK are declared in the system include file **WinSock2.inc**.

The Third Argument

The third argument passed to the **ioctlsocket** function is of type *u_long* and is passed by reference. You use this argument to further specify what the command should do or to retrieve values returned by the command.

Return Values

The **ioctlsocket** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file

WinSock2.inc.

Example

The following code fragments illustrates how to enable socket's non-blocking mode.

```
ulNonBlockingMode := 1;
iRet := ioctlsocket(s, FIONBIO, ulNonBlockingMode);
```

The code fragment above assumes that **s** is a socket that has already been created, **ulNonBlockingMode** is a variable of type *u_long*, and **iRet** is a variable of integer type. **NOTE:** The type *u_long* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.9.3 The select function

Description

The **select** function checks whether one or more sockets are *readable* or *writeable*, and/or checks one or more sockets for errors.

A socket is *readable* if:

- it is a data socket and data is available for reading (therefore the recv or recvfrom functions will not block).
- it is a data stream socket and a request to close the connection has been received from the socket at the other end of the connecion (therefore the recv function will return immedietly and not block).
- it is a listening socket and a connection request is pending (therefore the accept function will not block).

A socket is *writeable* if:

- it is a data socket and data can be sent (therefore the send or sendto functions will not block).
- the socket's non-blocking mode is enabled and a connection request from this socket has been accepted. **NOTE:** If you call the connect function on a socket, and the socket has it's non-blocking mode enabled, then connect will not wait for the connection request to be accepted by the socket at the other end. Instead the connect function will return *SOCKET_ERROR* immediately, as if the call failed, however the connection request will still be pending (the request will be queued by the listening socket at the other end). The socket will become *writeable* if the connection request is accepted.

The errors that the **select** function can detect are:

- The connect function was called on a socket, and the socket has it's non-blocking mode disabled, but the connection request was not accepted.
- OOB data is available from a socket but the SO_OOBINLINE socket option is disabled.

Declaration

The system include file **WinSock2.inc** contains the following declarations for the **select** function, and the *fd_set* and *timeval* types:

```
FD_SETLAST = 63;

fd_set = packed record
fd_count : u_int;        (* Number of sockets in the array *)
fd_array : array[0..FD_SETLAST] of SOCKET;      (* array of sockets *)
end;

timeval = packed record
tv_sec : integer;        (* seconds *)
tv_usec : integer;       (* and micro-seconds (i.e. 1/1,000,000 of a second) *)
end;

function select(nfds : integer; var readfds, writefds, exceptfds : fd_set;
var tmeout : timeval) : integer;
```

Arguments

At least one of the second, third, or fourth arguments must not be null.

The First Argument

The first argument is provided only for compatiblility with the Berkeley Sockets API and is ignored.

The Second Argument

The second argument to the **select** function is of type *fd_set*, and is passed by reference. When you call the **select** function, this argument should contain the set of sockets you want to check for *readability*. If you don't want to check any sockets for *readability* then use null as the value of this argument. When the **select** function returns, this argument contains the set of sockets that were checked and found to be *readable*.

The Third Argument

The third argument to the **select** function is of type *fd_set*, and is passed by reference. When you call the **select** function, this argument should contain the set of sockets you want to check for *writeability*. If you don't want to check any sockets for *writeability* then use null as the value of this argument. When the **select** function returns, this argument contains the set of sockets that were checked and found to be *writeable*.

The Fourth Argument

The fourth argument to the **select** function is of type *fd_set*, and is passed by reference. When you call the **select** function, this argument should contain the set of sockets you want to check for errors. If you don't want to check any sockets for errors then use null as the value of this argument. When the **select** function returns, this argument contains the set of sockets that were checked and found to have experienced errors.

The Fifth Argument

The fifth argument to the **select** function is of type *timeval*, and although it is passed by reference it is not modified by the call. This argument specifies the maximum length of time the **select** function should wait for one or more of checks on the sockets to succeed. If this argument is null then the **select** function will wait indefinitely.

Return Values

The **select** function returns the total number of sockets returned in the second, third, and fourth arguments is the call is successful. If the call fails then the value *SOCKET_ERROR* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Example

The following function uses the **select** function to check whether a socket is readable.

```
function IsSocketReadable(s : SOCKET; iMaxWaitSecs, iMaxWaitUSecs : integer) :
boolean;
var
s_set : fd_set;
tv : timeval;
iRet : integer;
begin (* IsSocketReadable *)
(* Put socket in the set of sockets to be tested *)
s_set.fd_count := 1;
s_set.fd_array[0] := s;

(* Specify the maximum wait time *)
tv.tv_sec := iMaxWaitSecs;
tv.tv_usec := iMaxWaitUSecs;

(* Check whether socket is readable *)
iRet := select(0, s_set, null, null, tv);
IsSocketReadable := (iret = 1);
end; (* IsSocketReadable *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.10.1 The accept function

Description

The **accept** function allows a *listening socket* to establish a connection. The underlying network service provider will queue-up requests to connect to the listening socket, and this function will extract the first connection request from the queue. You can specify the maximum size of the connection request queue when you put the socket in *listening* mode (i.e. when you call the listen function).

- Non-blocking mode disabled - If the *listening socket* has it's non-blocking mode **disabled** (which is the default condition), and no connection request is in the queue then this function will block, and wait until a connection request is made before returning to the caller.
- Non-blocking mode enabled - If the *listening socket* has it's non-blocking mode **enabled**, and no connection request is in the queue then this function will return *INVALID_SOCKET* indicating failure. If you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEWOULDBLOCK*. **NOTE:** The constants *WSAEWOULDBLOCK* and *INVALID_SOCKET* are declared in the system include file **WinSock2.inc**.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **accept** function:

```
function accept(s : SOCKET; addr : address; var addrlen : integer) : SOCKET;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **accept** function is the listening socket that you want to accept a connection request.

The Second Argument

The second argument passed to the **accept** function can be used to retrieve the *name* of the socket whose connection request was accepted.

If you are interested in the *name* of the socket whose connection request was accepted, then you should use the built-in function addr to pass the operating system address of a variable into which the **accept** function should store the socket's *name*. See the bind function for more information on socket names.

If you are not interested in the *name* of the socket whose connection request was accepted, then you should use null for this argument.

The Third Argument

The third argument passed to the **accept** function is a variable of integer type, and is passed by reference. When you call the **accept** function, the value stored in this argument is either the size of the variable that was passed in the second argument, or null if no variable was passed in the second argument. If this argument is not null then when the **accept** function returns, this argument contains the actual length of the *name* stored in the second argument.

Return Values

The **accept** function returns a new socket that is connected to the socket whose connection request was accepted, if the call is successful. If the call fails then the value *INVALID_SOCKET* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** *INVALID_SOCKET* is declared in the system include file **WinSock2.inc**.

After a connection request is accepted, the listening socket continues to be available to accept more connections, and the new connected socket should be used to send data to and receive data from the socket that requested the connection.

Example

The following code fragment shows one way to accept connections.

```
ConnectedSocket := accept(ListeningSocket, null, null);
if ConnectedSocket <> SOCKET_ERROR then
begin
...
use the connection to send and receive data
...
end
else
begin
...
handle the error
...
end
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.11.1 The connect function

Description

The action performed by the **connect** function depends on whether it is used on connection-oriented sockets (e.g. stream sockets) or connectionless sockets (e.g. datagram sockets).

Connection-Oriented Sockets

When the **connect** function is used on a connection-oriented socket, it uses that socket to send a connection request to a *listening socket* which is listening for connection requests (see the listen function for more information about listening sockets). What happens next depends on whether the socket sending the connection request has it's non-blocking mode enabled or disabled (see dealing with blocking for more information about the non-blocking mode of sockets).

- Non-blocking mode disabled - If the requesting socket has it's non-blocking mode disabled then the **connect** function will block, and wait for the connection request to either be accepted, or be rejected, or to fail because of an error.
- Non-blocking mode enabled - If the requesting socket has it's non-blocking mode enabled then the **connect** function will return *SOCKET_ERROR* immediately as if the call failed, however the connection request is still pending, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEWOULDBLOCK*. You can then use

the select function to monitor the progress of the connection request. As long as a socket has a connection request pending, calling the **connect** function on that socket will fail. **NOTE:** The constants *WSAEWOULDBLOCK* and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

If the listening socket accepts the connection request then a communications link is established between the socket that requested the connection and the socket created by the accept function, and both of these sockets are ready to send and receive data. Any attempt to reconnect a socket that is already connected will fail, and the WSAGetLastError function will return *WSAEISCONN*. **NOTE:** The constant *WSAEISCONN* is declared in the system include file **WinSock2.inc**.

Connectionless Sockets

When the **connect** function is used on a connectionless socket, it does not send a connection request, instead the **connect** function will specify the *name* of destination socket or sockets for the send function, and the *name* of source socket or sockets for the recv function. The send function will send datagrams to the destination socket or sockets, and the recv function will silently discard any datagrams received from sockets other than the source socket or sockets.

Unbound Sockets

If the **connect** function is used on a unbound socket (see the bind function for more information), a unique *name* is assigned to the socket by the system, and the socket is marked as bound.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **connect** function:

```
function connect(s : SOCKET; name : address; namelen : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **connect** function is either the socket that you want to send a connection request from (connection-oriented sockets only), or the socket for which you want to specify a *name* for default destination and source sockets (connectionless sockets only).

The Second Argument

The second argument passed to the **connect** function is either the *name* of the *listening socket* that you want to send a connection request to (connection-oriented sockets only), or the *name* of the default destination and source sockets (connectionless sockets only).

The Third Argument

The third argument passed to the **connect** function is the length of the *name* passed in the second argument.

Return Values

The **connect** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail.

Example

The following example function is taken from the sample sockets programs distributed with Irie Pascal, and demonstrates how to connect a socket to a listening socket.

```
//PURPOSE: Connects a client data socket
//PARAMETER(s):
//    1. s    - The socket to be connected
//    2. port - The TCP/IP port to connect the socket to
//    3. a    - The IP address to connect the socket to
procedure ConnectDataSocket(s : socket; port : integer; a : in_addr);
var
sa : sockaddr_in;
iRet : integer;
begin (* ConnectDataSocket *)
fill(sa, 0);
sa.sin_family := AF_INET;
sa.sin_port := htonl(port) shr 16;
sa.sin_addr := a;
iRet := connect(s, addr(sa), sizeof(sa));
if iRet <> 0 then
FatalSocketError('Call to connect failed');
end; (* ConnectDataSocket *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.12.1 The recv function

Description

The **recv** function reads data from a socket, and can be used on connection-oriented sockets (e.g. stream sockets) or connectionless sockets (e.g. datagram sockets).

If there is no data available to be read from the socket then what happens next depends on whether the socket has it's non-blocking mode enabled or disabled (see dealing with blocking for more information about the non-blocking mode of sockets).

- Non-blocking mode disabled - If the socket has it's non-blocking mode disabled, and there is no data available to be read from the socket, then the **recv** function will block, and wait for data to become available to be read from the socket, or for the connection to be closed (connection-

oriented sockets only), or for the connection to be reset (connection-oriented sockets only).
- Non-blocking mode enabled - If the socket has it's non-blocking mode enabled, and there is no data available to be read from the socket, then the **recv** function will return *SOCKET_ERROR* immediately as if the call failed, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEWOULDBLOCK*. You can then use the select function to find out when data becomes available to be read from the socket. **NOTE:** The constants *WSAEWOULDBLOCK* and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

Connection-Oriented Sockets

When the **recv** function is used on connection-oriented sockets, it will read all of the data waiting to be read, up to the size of the buffer you supply. If the **recv** function is used on a socket that has been configured for in-line reception of OOB data and there is OOB data waiting to be read then only the OOB data will be read. Your program can use the ioctlsocket to determine whether there is OOB data waiting to be read.

If the socket at the other end of the connection has shut down the connection gracefully, and all of the data sent has been received, the **recv** function will return immediately with zero bytes received. If the connection has been reset, the **recv** function will return *SOCKET_ERROR*, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAECONNRESET*. **NOTE:** The constants *WSAECONNRESET* and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

Connectionless Sockets

When the **recv** function is used on connectionless sockets, it will read the first datagram waiting to be read. If the first datagram waiting to be read is larger than the buffer you supply, then only the first part of the datagram will be read (enough to fill the buffer), and the **recv** function call will return *SOCKET_ERROR* as if the call failed, and if you then call the WSAGetLastError function the error code returned will be *WSAEMSGSIZE*. For unreliable protocols (for example, UDP) the unread part of the datagram will be lost. For reliable protocols, the entire datagram is retained until it is successfully read by calling the **recv** function with a large enough buffer.

Before using the **recv** function on connectionless sockets the connect function must be called to specify the *name* of the source socket or sockets from which datagrams will be read. The **recv** function will silently discard datagrams from all sockets except the source socket or sockets.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **recv** function:

```
function recv(s : SOCKET; buf : address; len, flags : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **recv** function is the socket that you want to read data from.

The Second Argument

The second argument passed to the **recv** function is the operating system address of the buffer that you have supplied to store the data read from the socket. The type of this argument is address which means that you must use the built-in function addr to pass the operating system address of the buffer.

The Third Argument

The third argument passed to the **recv** function is the length of the buffer you have supplied to store the data read from the socket.

The Fourth Argument

The fourth argument passed to the **recv** function can specify flags which influence the behavior of the **recv** function. The flags can be OR'd or added together if more than one flag is being specified. The flags which can be used are:

- MSG_PEEK - Peeks at the data in the socket. The data is copied from the socket into the buffer supplied, but is not removed from the socket. When this flag is used the **recv** function returns the number of bytes of data waiting available to be read.
- MSG_OOB - Read OOB data from the socket.

If you do not wish to specify any flags then use zero as the value of this argument.

Return Values

The **recv** function returns the number of bytes read from the socket, if the call is successful. If the call fails the value *SOCKET_ERROR* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

If the socket on the other end of the connection has gracefully closed the connection then zero is returned (connection-oriented sockets only).

Example

The following function is taken from the sample program **wSpamFilter** and uses the **recv** function to read data from a socket. The function **SocketReadyForReading** is defined by the sample program and checks whether there is data waiting to be read from a socket. The type *LineType* is a *cstring* type.

```
//PURPOSE: Reads data from a socket
//PARAMETER(S):
//    1. buffer - used to store the data read
//    2. wMaxWait - maximum number of ms to wait for the socket to become ready.
//GLOBAL(s):
//    1. DataSocket - The socket to read from.
//NOTES:
//    The socket is checked to make sure it is ready for reading
//    before attempting to read. This makes it unlikely that the application
//    will become blocked waiting for data to come in.
procedure ReadDataSocket(var buffer : LineType; wMaxWait : word);
var
iDataRead : integer;
begin (* ReadDataSocket *)
```

```
buffer := '';
if SocketReadyForReading(DataSocket, wMaxWait) then
begin
writeln('Reading...');
LogMessage('Reading...');
iDataRead := recv(DataSocket, addr(buffer), sizeof(buffer)-2, 0);
if iDataRead=SOCKET_ERROR then
FatalSocketError('Call to recv failed')
else if iDataRead > 0 then
buffer[iDataRead+1] := chr(0);
end
end; (* ReadDataSocket *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.12.2 The recvfrom function

Description

The **recvfrom** function reads data from a socket and optionally retrieves the *name* of the socket that sent the data. The **recvfrom** function can be used on connection-oriented sockets (e.g. stream sockets) or connectionless sockets (e.g. datagram sockets). Before calling the **recvfrom** function to read data from a socket, a local *name* must have been assigned to the socket, this can be done explicitly with the bind function or implicitly with the sendto function.

If there is no data available to be read from the socket then what happens next depends on whether the socket has it's non-blocking mode enabled or disabled (see dealing with blocking for more information about the non-blocking mode of sockets).

- Non-blocking mode disabled - If the socket has it's non-blocking mode disabled, and there is no data available to be read from the socket, then the **recvfrom** function will block, and wait for data to become available to be read from the socket, or for the connection to be closed (connection-oriented sockets only), or for the connection to be reset (connection-oriented sockets only).
- Non-blocking mode enabled - If the socket has it's non-blocking mode enabled, and there is no data available to be read from the socket, then the **recvfrom** function will return *SOCKET_ERROR* immediately as if the call failed, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEWOULDBLOCK*. You can then use the select function to find out when data becomes available to be read from the socket. **NOTE:** The constants *WSAEWOULDBLOCK* and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

Connection-Oriented Sockets

When the **recvfrom** function is used on connection-oriented sockets, it will read all of the data waiting to be read, up to the size of the buffer you supply. If the **recvfrom** function is used on a socket that has been configured for in-line reception of OOB data and there is OOB data waiting to be read then only the OOB data will be read. Your program can use the ioctlsocket to determine whether there is OOB data waiting to be read.

If the socket at the other end of the connection has shut down the connection gracefully, and all of the data sent has been received, the **recvfrom** function will return immediately with zero bytes received. If the connection has been reset, the **recvfrom** function will return *SOCKET_ERROR*, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAECONNRESET*. **NOTE:** The constants *WSAECONNRESET* and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

Connectionless Sockets

When the **recvfrom** function is used on connectionless sockets, it will read the first datagram waiting to be read. If the first datagram waiting to be read is larger than the buffer you supply, then only the first part of the datagram will be read (enough to fill the buffer), and the **recvfrom** function call will return *SOCKET_ERROR* as if the call failed, and if you then call the WSAGetLastError function the error code returned will be *WSAEMSGSIZE*. For unreliable protocols (for example, UDP) the unread part of the datagram will be lost. For reliable protocols, the entire datagram is retained until it is successfully read by calling the **recvfrom** function with a large enough buffer.

The *name* of the socket that sent the data is retrieved if you supply a buffer to store the *name* (see the fifth and sixth arguments below).

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **recvfrom** function:

```
function recvfrom(s : SOCKET; buf : address; len, flags : integer;
from : address; var fromlen : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **recvfrom** function is the socket that you want to read data from.

The Second Argument

The second argument passed to the **recvfrom** function is the operating system address of the buffer that you have supplied to store the data read from the socket. The type of this argument is address which means that you must use the built-in function addr to pass the operating system address of the buffer.

The Third Argument

The third argument passed to the **recvfrom** function is the length of the buffer you have supplied to store the data read from the socket.

The Fourth Argument

The fourth argument passed to the **recvfrom** function can specify flags which influence the behavior of the **recvfrom** function. The flags can be OR'd or added together if more than one flag is being specified. The flags which can be used are:

- MSG_PEEK - Peeks at the data in the socket. The data is copied from the socket into the buffer supplied, but is not removed from the socket. When this flag is used the **recvfrom** function returns the number of bytes of data waiting available to be read.
- MSG_OOB - Read OOB data from the socket.

If you do not wish to specify any flags then use zero as the value of this argument.

The Fifth Argument

The fifth argument passed to the **recvfrom** function is optional and should either be zero or the operating system address of a buffer to store the *name* of the socket that sent the data. If you are passing the operating address of a buffer then you should use the built-in function addr to get this address. This argument is ignored when reading from connection-oriented sockets.

The Sixth Argument

The sixth argument passed to the **recvfrom** function is the length of the buffer you supplied to store the *name* of the socket that sent the data. If you supply a buffer to store the *name* then after the **recvfrom** function returns successfully this argument will contain the actual length of the *name* stored in the buffer. This argument is ignored when reading from connection-oriented sockets.

Return Values

The **recvfrom** function returns the number of bytes read from the socket, if the call is successful. If the call fails the value *SOCKET_ERROR* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail.
NOTE: The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

If the socket on the other end of the connection has gracefully closed the connection then zero is returned (connection-oriented sockets only).

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.13.1 The send function

Description

The **send** function sends data through a socket, and can be used on connection-oriented sockets (e.g. stream sockets) or connectionless sockets (e.g. datagram sockets). The behavior of this function may depend on whether the socket has it's non-blocking mode enabled or disabled (see dealing with blocking for more information about the non-blocking mode of sockets).

- Non-blocking mode disabled - If the socket has it's non-blocking mode disabled, and there is no buffer space available within the transport system to store the data to be transmitted, then the **send** function will block, and wait for buffer space to become available, or for the connection to

be closed (connection-oriented sockets only), or for the connection to be reset (connection-oriented sockets only).
- Non-blocking mode enabled - If the socket has it's non-blocking mode enabled, then the number of bytes actually sent can be less than the amount you are trying to send, depending the availability of buffer space within the transport system to store the data to be transmitted. If there is no buffer space available within the transport system to store the data to be transmitted, then the **send** function will fail and return *SOCKET_ERROR*, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEWOULDBLOCK*. You can use the select function to find out when some buffer space becomes available. **NOTE:** The constants *WSAEWOULDBLOCK* and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

Connectionless Sockets

When the **send** function is used on connectionless sockets, care must be taken not to exceed the maximum packet size of the underlying provider, which can be obtained by using getsockopt to retrieve the value of socket option *SO_MAX_MSG_SIZE*. If the data is too long to pass atomically through the underlying protocol, the error *WSAEMSGSIZE* is returned, and no data is transmitted.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **send** function:

```
function send(s : SOCKET; buf : address; len, flags : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **send** function is the socket that you want to send data through.

The Second Argument

The second argument passed to the **send** function is the operating system address of the buffer containing the data to be sent through the socket. The type of this argument is address which means that you must use the built-in function addr to pass the operating system address of the buffer.

The Third Argument

The third argument passed to the **send** function is the number of bytes of data to be sent through the socket. This argument can be zero, and this case will be treated by implementations as successful, and send will return zero as a valid value. If the socket is connectionless, a zero-length transport datagram is sent.

The Fourth Argument

The fourth argument passed to the **send** function can specify flags which influence the behavior of the

send function. The flags can be OR'd or added together if more than one flag is being specified. The flags which can be used are:

- MSG_DONTROUTE - Specifies that the data sent should not be subject to routing. This flag may be ignored.
- MSG_OOB - Sends OOB data through the socket.

If you do not wish to specify any flags then use zero as the value of this argument.

Return Values

The **send** function returns the number of bytes sent through the socket, if the call is successful. If the call fails the value *SOCKET_ERROR* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

The number of bytes sent through the socket can be less than the number of bytes you are trying to send. Also the fact that data has been successfully sent does not guarantee that the data was successfully received at the other end.

Example

The following procedure is taken from the sample program **wSpamFilter** and uses the **send** function to send data through a socket. The type *LineType* is a *cstring* type. A CR/LF pair is appended to the end of the data because the server at the other end expects it, and is not a requirement for using the **send** function.

```
//PURPOSE: Writes and logs a line to the data socket.
//PARAMETER(s):
//    1. buffer - Contains the data to write to the socket.
//NOTES:
//    A CR/LF pair is appended to the line before writing to the socket.
procedure WriteDataSocket(var buffer : LineType);
var
iRet : integer;
begin (* WriteDataSocket *)
writeln('Writing ', buffer);
LogMessage('OUT ' + buffer);
buffer := buffer + CR + LF;
iRet := send(DataSocket, addr(buffer), length(buffer), 0);
end; (* WriteDataSocket *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.13.2 The sendto function

Description

The **sendto** function sends data through a socket, and can be used on connection-oriented sockets (e.g.

stream sockets) or connectionless sockets (e.g. datagram sockets). The behavior of this function may depend on whether the socket has it's non-blocking mode enabled or disabled (see dealing with blocking for more information about the non-blocking mode of sockets).

- Non-blocking mode disabled - If the socket has it's non-blocking mode disabled, and there is no buffer space available within the transport system to store the data to be transmitted, then the **sendto** function will block, and wait for buffer space to become available, or for the connection to be closed (connection-oriented sockets only), or for the connection to be reset (connection-oriented sockets only).
- Non-blocking mode enabled - If the socket has it's non-blocking mode enabled, then the number of bytes actually sent can be less than the amount you are trying to send, depending the availability of buffer space within the transport system to store the data to be transmitted. If there is no buffer space available within the transport system to store the data to be transmitted, then the **sendto** function will fail, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEWOULDBLOCK*. You can use the select function to find out when some buffer space becomes available. **NOTE:** The constant *WSAEWOULDBLOCK* is declared in the system include file **WinSock2.inc**.

If no local *name* has been assigned to the socket, then a unique local *name* is assigned to the socket by the system. Your program can use the getsockname function to determine the local *name* assigned to the socket.

Connectionless Sockets

When the **sendto** function is used on connectionless sockets, care must be taken not to exceed the maximum packet size of the underlying provider, which can be obtained by using the getsockopt function to retrieve the value of socket option *SO_MAX_MSG_SIZE*. If the data is too long to pass atomically through the underlying protocol, the **sendto** function will fail, and if you then call the WSAGetLastError function to find out the reason for the error *WSAEMSGSIZE* is returned, and no data is transmitted.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **sendto** function:

```
function sendto(s : SOCKET; buf : address; len, flags : integer;
toaddr : address; tolen : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **sendto** function is the socket that you want to send data through.

The Second Argument

The second argument passed to the **sendto** function is the operating system address of the buffer containing the data to be sent through the socket. The type of this argument is address which means that you must use the built-in function addr to pass the operating system address of the buffer.

The Third Argument

The third argument passed to the **sendto** function is the number of bytes of data to be sent through the socket. This argument can be zero, and this case will be treated by implementations as successful, and send will return zero as a valid value. If the socket is connectionless, a zero-length transport datagram is sent.

The Fourth Argument

The fourth argument passed to the **sendto** function can specify flags which influence the behavior of the **send** function. The flags can be OR'd or added together if more than one flag is being specified. The flags which can be used are:

- MSG_DONTROUTE - Specifies that the data sent should not be subject to routing. This flag may be ignored.
- MSG_OOB - Sends OOB data through the socket.

If you do not wish to specify any flags then use zero as the value of this argument.

The Fifth Argument

The fifth argument passed to the **sendto** function usually specifies the *name* of the destination socket or sockets that should receive the data being sent. If the connect function has been used previously on the socket that you are sending the data through, to specify a default destination socket or default destination sockets then:

- this argument can be zero (if you want to send the data to the default destination socket or sockets)
- this argument can override the default destination socket or sockets, for the current datagram only, and specify the *name* of a destination socket, or destination sockets that should receive the data being sent.

The *name* specified by this argument, can be any valid *name* in the socket's address family, including a *name* with broadcast or multicast addresses. To send data to a broadcast address your program must have previously used the setsockopt function with *SO_BROADCAST* enabled. Otherwise, the **sendto** function will fail, and if you then call the WSAGetLastError function to find out the reason for the failure, the error code returned will be *WSAEACCES*. For TCP/IP sockets, your program can send to any multicast address.

The following rules apply when sending broadcast datagrams through sockets from the Internet Address Family:

- To send a broadcast (through SOCK_DGRAM sockets only), the address in this argument should be the special IP address *INADDR_BROADCAST*.
- Broadcast datagrams should not exceed the size at which fragmentation can occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

This argument is ignored when sending data through connection-oriented sockets.

The Sixth Argument

The sixth argument passed to the **sendto** function is the length of the *name* of the destination socket or sockets, that was specified by the fifth argument. This argument is ignored when sending data through connection-oriented sockets.

Return Values

The **sendto** function returns the number of bytes sent through the socket, if the call is successful. If the call fails the value *SOCKET_ERROR* is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail.
NOTE: The constants *SOCKET_ERROR*, *WSAEACCES*, *SO_BROADCAST*, *MSG_DONTROUTE*, *MSG_OOB*, *WSAEMSGSIZE*, and *INADDR_BROADCAST* are declared in the system include file **WinSock2.inc**.

The number of bytes sent through the socket can be less than the number of bytes you are trying to send. Also the fact that data has been successfully sent does not guarantee that the data was successfully received.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.14.1 The gethostbyname function

Description

The **gethostbyname** function returns information about a host identified by name. The host information is returned in a structure of type **hostent**. This structure is allocated by the WinSock2 library, and your program should not attempt to modify or de-allocate, the structure or any of it's components.

Only one copy of the **hostent** structure is allocated per calling thread, and since Irie Pascal programs only have one thread, there will be only one copy of this structure allocated for your program. As a result, you should copy any host information you need from the structure before calling any other Windows Sockets functions.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **hostent** and **p_hostent** types and the **gethostbyname** function:

```
hostent = packed record
h_name : address;
h_aliases : address;
h_addrtype : shortint;
h_length : shortint;
h_addr_list : address;
end;

p_hostent = ^hostent;

function gethostbyname(name : address) : p_hostent;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first, and only, argument passed to the **gethostbyname** function is the operating system address of the buffer containing the name of the host you want to get information about. The type of this argument is the built-in type address, so you must use the built-in function addr to get the operating system address of the buffer. The host name contained in the buffer must be a null-terminated string (i.e. a *cstring*).

The buffer pointed to by this argument must contain a valid host name and not an address. If you have the address (as a null-terminated string) of a host instead of the name of the host, and you want to get information about that host then you should do the following:

1. Use the inet_addr function to convert the address, represented as a string into an actual address (in numeric format).
2. Use the gethostbyaddr function to get the host information.

Return Values

The **gethostbyname** function returns a pointer to the **hostent** structure that contains the host information, if the call is successful. If the call fails then the value nil is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail.

Example

The following procedure was taken from one of the sample programs distributed with Irie Pascal and shows one way to use the **gethostbyname** function.

```
//PURPOSE: Given the name of a host performs a DNS lookup
//PARAMETER(s):
//    1. sName - contains the name of the host
procedure DNSLookUp(strName : string);
var
pHostEnt : p_hostent;
cstrName : cstring;
begin
//Convert the name to a cstring since 'gethostbyname' expects a cstring
cstrName := strName;
pHostEnt := gethostbyname(addr(cstrName));
if pHostEnt = nil then
writeln('Can not find host ''', cstrName, '''')
else
DisplayHostEntInfo(pHostEnt);
end;
```

The procedure **DisplayHostEntInfo** is defined by the sample program and displays the host information stored in the **hostent** structure. The procedure **DisplayHostEntInfo** is below and shows one way to access the host information.

```
//PURPOSE: Displays the information that was obtained from a DNS or Reverse DNS
lookup
//PARAMETER(s):
//    1. pHostEnt - a pointer to the sockets API hostent structure that contains
//       the result of the lookup.
//NOTES:
//    This procedure uses the Windows API functions 'lstrlen' and 'lstrcpy' to
```

```
//manipulate strings in C format (i.e. null-terminated arrays of char). These
//functions are used because the strings in the hostent structure are in C format.
//Since these functions are not described in the online help (because they are
//Windows API functions not Irie Pascal functions) a brief description is given
//below for persons not already familiar with these functions:
// 'lstrlen' returns the length of the string (not counting the null terminator).
// 'lstrcpy' copies the string pointed to by it's second argument into the memory
//   pointed to by it's first argument (the null terminator is also copied).
procedure DisplayHostEntInfo(pHostEnt : p_hostent);
const
MAX_ASCII_ADDRESS = 15;
var
AddrTemp, AddrAddr, AddrRet : address;
iRet : integer;
strAddress : cstring[MAX_ASCII_ADDRESS];
strName : cstring;

//PURPOSE: Returns the address pointed to be another address.
//PARAMETER(s):
//    1. a - an address pointing to another address
//RETURNS:
//    The address pointed to by the parameter 'a'.
//NOTES:
//Irie Pascal pointer variables contain virtual addresses that are only
//meaningful to the Irie Pascal Run Time Engine. However when using the
//Windows API you occassionally need to manipulate non-virtual addresses.
//To make this easier Irie Pascal supports a new type called 'address'.
//However you can't use ^ to dereference an address, because addresses
//are like untyped pointers (i.e. the compiler does not know the type of
//the value pointed to by the address, so it can not generate code to return
//the value pointed to be the address). The address type is assignment
//compatible with all pointer types and vice/versa, so the solution to this
//problem is to assign the address to suitable pointer variable and then
//dereference the pointer variable.
//In the case of the function below the address is known to be pointing
// to another address. So the address to be deferenced is assigned to
// a pointer to an address, and then this pointer is dereferenced.
function DereferenceAddress(a : address) : address;
var
p : ^address;
begin
p := a;
DereferenceAddress := p^;
end;

begin (* DisplayHostEntInfo *)
//pHostEnt^.h_name is the address of a null-terminated string containing
//the name of the host.
writeln('NAME:');
if pHostEnt^.h_name=NULL then
writeln('NOT FOUND')
else if (lstrlen(pHostEnt^.h_name)+1) > sizeof(strName) then
writeln('TOO LONG')
else
begin
iRet := lstrcpy(addr(strName), pHostEnt^.h_name);
writeln(strName);
end;
writeln;

//pHostEnt^.h_aliases is the address of a null terminated array of addresses
//of null-terminated strings containing alternative names for the host.
writeln('ALIASES:');
AddrAddr := pHostEnt^.h_aliases;
if AddrAddr = NULL then
writeln('None')
else
begin
AddrTemp := DereferenceAddress(AddrAddr);
```

```
while AddrTemp <> NULL do
begin
if lstrlen(AddrTemp) > 0 then
begin
if (lstrlen(addrTemp)+1) > sizeof(strName) then
writeln('TOO LONG')
else
begin
iRet := lstrcpy(addr(strName), AddrTemp);
writeln(strName);
end;
end
else
writeln('EMPTY');

AddrAddr := AddrAddr + sizeof(address);
AddrTemp := DereferenceAddress(AddrAddr);
end;
end;
writeln;

if pHostEnt^.h_addrtype <> AF_INET then
writeln('Invalid address type')
else if pHostEnt^.h_length <> sizeof(address) then
writeln('Invalid address length')
else
begin
//pHostEnt^.h_addr_list is the address of a null terminated array of
//addresses of IP addresses of the host.
writeln('ADDRESSES:');
AddrAddr := pHostEnt^.h_addr_list;
if AddrAddr = NULL then
writeln('None')
else
begin
//Get the first element of the array
AddrTemp := DereferenceAddress(AddrAddr);
while AddrTemp <> NULL do
begin
//Dereference the current array element to get the
//IP address.
AddrTemp := DereferenceAddress(AddrTemp);
//Convert the IP address from binary format to a human
//readable format (like nnnn.nnnn.nnnn.nnn)
AddrRet := inet_ntoa(AddrTemp);
if (AddrRet=null) or ((lstrlen(AddrRet)+1)>sizeof(strAddress)) then
writeln('[ERROR]')
else
begin
iRet := lstrcpy(addr(strAddress), AddrRet);
writeln(strAddress);
end;

AddrAddr := AddrAddr + sizeof(address);
//Get the next element of the array
AddrTemp := DereferenceAddress(AddrAddr);
end;
end;
writeln;
end;
end; (* DisplayHostEntInfo *)
```

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.14.2 The gethostbyaddr function

Description

The **gethostbyaddr** function returns information about a host identified by address. The host information is returned in a structure of type **hostent**. This structure is allocated by the WinSock2 library, and your program should not attempt to modify or de-allocate, the structure or any of it's components.

Only one copy of the **hostent** structure is allocated per calling thread, and since Irie Pascal programs only have one thread, there will be only one copy of this structure allocated for your program. As a result, you should copy any host information you need from the structure before calling any other Windows Sockets functions.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **hostent** and **p_hostent** types and the **gethostbyaddr** function:

```
hostent = packed record
h_name : address;
h_aliases : address;
h_addrtype : shortint;
h_length : shortint;
h_addr_list : address;
end;

p_hostent = ^hostent;

function gethostbyaddr(addr : address; len, typ : integer) : address;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **gethostbyaddr** function is the operating system address of the buffer containing the address of the host you want to get information about. The address stored in the buffer is not stored as a string, it is stored as a numeric value in in network byte order. The type of this argument is the built-in type address, so you must use the built-in function addr to get the operating system address of the buffer.

If you have the address of a host, as a null-terminated string, and you want to get information about that host then you should do the following:

1. Use the inet_addr function to convert the address, represented as a string into an address, represented in a numeric format.
2. Use the gethostbyaddr function to get the host information.

The Second Argument

The second argument passed to the **gethostbyaddr** function is the length of the host address in the buffer pointed to be the first argument.

The Third Argument

The third argument passed to the **gethostbyaddr** function is the type of the host address in the buffer pointed to be the first argument.

Return Values

The **gethostbyaddr** function returns a pointer to the **hostent** structure that contains the host information, if the call is successful. If the call fails then the value nil is returned, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail.

Example

The following procedure was taken from one of the sample programs distributed with Irie Pascal and shows one way to use the **gethostbyaddr** function.

```
//PUPOSE: Given the IP address of a host perform a reverse DNS lookup.
//    1. strIPAddress - contains the name of the host
procedure ReverseDNSLookUp(strIPAddress : cstring);
var
IPAddress : in_addr;
pHostEnt : p_hostent;
begin
IPAddress := inet_addr(addr(strIPAddress));
if IPAddress = INADDR_NONE then
writeln('Invalid IP address')
else
begin
pHostEnt := gethostbyaddr(addr(IPAddress), sizeof(IPAddress), AF_INET);
if pHostEnt = nil then
writeln('Can not find address ''', hex(IPAddress), '''')
else
DisplayHostEntInfo(pHostEnt);
end;
end;
```

The procedure **DisplayHostEntInfo** is defined by the sample program and displays the host information stored in the **hostent** structure. See the gethostbyname function for the actual **DisplayHostEntInfo** procedure.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.15.1 The shutdown function

Description

The **shutdown** function disables the reception of data from a socket, and/or the transmission of data through a socket, and can be used on all types of sockets. The **shutdown** function does **not** close the socket. Any resources attached to the socket will not be freed until the closesocket function is called.

To ensure that all data is sent and received on a connected socket before it is closed, your program should use the **shutdown** function to close the connection before calling the closesocket function to close the socket. For example, to initiate a graceful disconnect:

- Call the **shutdown** to disable the transmission of data.
- Call the recv function until zero or *SOCKET_ERROR* is returned.
- Call the closesocket function.

The **shutdown** function does not block regardless of the *SO_LINGER* setting on the socket. You should not rely on being able to re-use a socket after it has been shut down. In particular, a Windows Sockets provider is not required to support the use of the connect function on a socket that has been shut down.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **shutdown** function:

```
function shutdown(s : SOCKET; how : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **shutdown** function is the socket that you want to disable, the reception of data from and/or disable the transmission of data through.

The Second Argument

The second argument passed to the **shutdown** function specifies whether you want to disable the receptioin of data, the transmission of data or both, and can have the follow values:

- SD_RECEIVE - subsequent calls to the recv function on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP sockets, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.
- SD_SEND - subsequent calls to the send function are disallowed. For TCP sockets, a *FIN* will be sent after all data is sent and acknowledged by the receiver.
- SD_BOTH - disables both sending and receiving as described above.

Return Values

The **shutdown** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constants *SO_LINGER*, *SD_RECEIVE*, *SD_SEND*, *SD_BOTH*, and *SOCKET_ERROR* are declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.15.2 The closesocket function

Description

The **closesocket** function closes a socket, and frees the resources attached to the socket. To assure that all data is sent and received on a connection, your program should call the shutdown function before calling the **closesocket** function. **NOTE**: An *FD_CLOSE* network event is not posted after the **closesocket** function is called.

Your program should always have a matching call to the **closesocket** function for each successful call to the createsocket function to return any socket resources to the system.

The behavior of the **closesocket** function can be affected by the socket options *SO_LINGER* and *SO_DONTLINGER*. When a socket is created *SO_DONTLINGER* is enabled (and therefore *SO_LINGER* is disabled).

SO_LINGER Enabled

If the *SO_LINGER* socket option is enabled with a zero time-out interval (that is, the *linger* structure members *l_onoff* is not zero and *l_linger* is zero), the **closesocket** function will not block even if queued data has not yet been sent or acknowledged. This is called a hard or abortive close, because the connection is reset immediately, and any unsent data is lost. Any recv function call on the remote side of the connection will fail, and the WSAGetLastError function with return *WSAECONNRESET*.

If the *SO_LINGER* socket option is set with a nonzero time-out interval on a socket with it's non-blocking mode disabled, the **closesocket** function will block until any remaining data has been sent or until the time-out expires. This is called a graceful disconnect. If the time-out expires before all data has been sent, the Windows Sockets implementation terminates the connection before the **closesocket** function returns.

Enabling the *SO_LINGER* socket option with a nonzero time-out interval on a socket with it's non-blocking mode enabled is not recommended. In this case, the call to the **closesocket** function will fail, and the WSAGetLastError function will return *WSAEWOULDBLOCK* if the close operation cannot be completed immediately. However the socket handle is still valid, and a disconnect is not initiated. Your program must call the **closesocket** function again to close the socket.

SO_DONTLINGER Enabled

If the *SO_DONTLINGER* socket option is enabled on a stream socket by setting the *l_onoff* member of the *linger* structure to zero, the **closesocket** function call will return immediately and successfully, whether the socket has it's non-blocking mode enabled or disabled. However, any data queued for transmission will be sent, if possible, before the underlying socket is closed. This is also called a graceful disconnect. In this case, the Windows Sockets provider cannot release the socket and other resources for an arbitrary period, thus affecting applications that expect to use all available sockets. This is the default behavior (*SO_DONTLINGER* is enabled by default).

Declaration

The system include file **WinSock2.inc** contains the following declarations for the *linger* type, and the **closesocket** function:

```
linger = packed record
l_onoff : u_short;
l_linger : u_short;
end;

function closesocket(s : SOCKET) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first and only argument passed to the **closesocket** function is the socket that you want to close.

Return Values

The **closesocket** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constants *SO_LINGER*, *SO_DONTLINGER*, *WSAECONNRESET*, *WSAEWOULDBLOCK*, *SOCKET_ERROR*, and the type *linger* are declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.1 The gethostname function

Description

The **gethostname** function retrieves the host name of the local computer. This host name can be a simple host name or a fully qualified domain name, but whatever it's form the host name retrieved is guaranteed to be successfully parsed by the gethostbyname function. This is true even if no host name has been configured for the local computer.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **gethostname** function:

```
function gethostname(name : address; namelen : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **gethostname** function is the operating system address of the buffer that you supply to store the host name retrieved. The type of this argument is the built-in type address, so you must use the built-in function addr to get the operating system address of the buffer. The host name that will be put into this buffer by the **gethostname** function is a null-terminated string (i.e. a *cstring*).

The Second Argument

The second argument passed to the **gethostname** function, is the length of the buffer you supplied to store the host name.

Return Values

The **gethostname** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.2 The getpeername function

Description

The **getpeername** function retrieves the *name* of a socket's peer (i.e. the *name* of the socket at the other end of the connection). **NOTE:** In the case of a connectionless socket, the *name* retrieved will be the *name* of the default source or destination sockets, specified by a previous call to the connect function. The *names* of sockets specified in previous calls to the sendto function will not be retrieved by this function.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **getpeername** function:

```
function getpeername(s : SOCKET; name : address; var namelen : integer) : integer;
external dll='wsock32.dll';
```

Arguments

The First Argument

The first argument passed to the **getpeername** function is the socket whose peer you want to get the name of.

The Second Argument

The second argument passed to the **getpeername** function is the operating system address of a buffer to store the *name* of the socket at the other end of the connection. You should use the built-in function addr to get the operating system address of this buffer.

The Third Argument

The third argument passed to the **getpeername** function is the length of the buffer you supplied to store the *name* of the socket at the other end of the connection. After the **getpeername** function returns successfully this argument will contain the actual length of the *name* stored in the buffer.

Return Values

The **getpeername** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.3 The getsockname function

Description

The **getsockname** function retrieves a socket's *name*, and can be useful in cases where you have not explicity assigned a *name* to the socket, and the system has assigned a *name* for you. One such case occurs, if you use the connection function on a socket without first assigning a *name* to the socket with the bind function.

For connection-oriented sockets, you should not assume that the socket's address will available, if the socket has been bound to an unspecified address (using *ADDR_ANY*), and the socket is not connected.

For connectionless sockets, the address may not be available unil I/O occurs on the socket.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **getsockname** function:

```
function getsockname(s : SOCKET; name : address; var namelen : integer) : integer;
external dll='wsock32.dll';
```

Arguments

The First Argument

The first argument passed to the **getsockname** function is the socket whose *name* you want to get.

The Second Argument

The second argument passed to the **getsockname** function is the operating system address of a buffer you have supplied to store the *name* of the socket. You should use the built-in function addr to get the operating system address of this buffer.

The Third Argument

The third argument passed to the **getsockname** function is the length of the buffer you supplied to store the *name* of the socket. After the **getsockname** function returns successfully this argument will contain the actual length of the *name* stored in the buffer.

Return Values

The **getsockname** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.4 The getsockopt function

Description

The **getsockopt** function retrieves the current value of a socket's option. If an option has not been set with the setsockopt function, then the **getsockopt** function returns the default value for the option.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **getsockopt** function:

```
function getsockopt(s : SOCKET; level, optname : integer; optval : address; var
optlen : integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **getsockopt** function is the socket from which you want to get the option value.

The Second Argument

The second argument passed to the **getsockopt** function is the level at which the socket option is defined. You will usually use *SOL_SOCKET* as the value of this argument.

The Third Argument

The third argument passed to the **getsockopt** function is an integer value that identifies the socket option whose value you want to retrieve. The socket options you are most likely to retrieve are given below:

- SO_ACCEPTCONN - This option is of type **BOOL**, and indicates whether the socket is listening.
- SO_BROADCAST - This option is of type **BOOL**, and indicates whether you can use the socket to send broadcast messages.
- SO_DEBUG - This option is of type **BOOL**, and indicates whether debugging is enabled.
- SO_DONTLINGER - This option is of type **BOOL**, and indicates whether the *SO_LINGER* option is disabled.
- SO_DONTROUTE - This option is of type **BOOL**, and indicates whether routine is disabled.
- SO_ERROR - This option is of integer type, and is the error code stored for the socket. The system stores an error code for each socket, which is different from the error code returned by the WSAGetLastError function, which is stored for each thread. Calling the **getsockopt** function to retrieve the error code stored for a socket causes the error code to be reset. However a successful call to a WinSock2 library function, does not reset the error code stored for each socket.
- SO_LINGER - This opton is of type *linger* and returns the current linger options for the socket (see the closesocket function for more information about linger options).
- SO_MAX_MSG_SIZE - This option is of integer type, and is the maximum size of a datagram for datagram sockets.
- SO_OOBINLINE - This option is of type **BOOL**, and indicates whether OOB data will be received in the normal data stream.
- SO_TYPE - This option is of integer type, and is the kind of the socket (e.g. *SOCK_STREAM*.

NOTE: Socket options of type **BOOL** are retrieved into an integer, and a value of zero means **false**, and an non-zero value means **true**.

The Fourth Argument

The fourth argument passed to the **getsockopt** function is the operating system address of the buffer you have supplued to store the value of the socket option. You should use the built-in function addr to get the operating system address of this buffer. For the *SO_LINGER* socket option, the buffer should be a *linger* structure, for the other options the buffer should be of integer type.

The Fifth Argument

The fifth argument passed to the **getsockopt** function is the length of the buffer you supplied to store the value of the socket option. After the **getsockopt** function returns successfully this argument will contain the actual length of the socket option value stored in the buffer.

Return Values

The **getsockopt** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.5 The htonl function

Description

The **htonl** function converts a value of type *u_long* from host to TCP/IP network byte order (which is big-endian).

Declaration

The system include file **WinSock2.inc** contains the following declarations for the *u_long* type, and the **htonl** function:

```
u_long = word;

function htonl(hostlong : u_long) : u_long;
external dll='ws2_32.dll';
```

Arguments

The only argument passed to the **htonl** function is the *u_long* value that you want to convert to TCP/IP network byte order.

Return Values

The **htonl** function returns the *u_long* value in TCP/IP network byte order.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.6 The htons function

Description

The **htons** function converts a value of type *u_short* from host to TCP/IP network byte order (which is big-endian).

Declaration

The system include file **WinSock2.inc** contains the following declarations for the *u_short* type, and the **htonl** function:

```
u_short = shortint;

function htonl(hostlong : u_long) : u_long;
external dll='ws2_32.dll';
```

Arguments

The only argument passed to the **htons** function is the *u_short* value that you want to convert to TCP/IP network byte order.

Return Values

The **htons** function returns the *u_short* value in TCP/IP network byte order.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.7 The inet_addr function

Description

The **inet_addr** function converts a host address from a null-terminated string into a numeric value.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **inet_addr** function:

```
function inet_addr(cp : address) : u_long;
external dll='ws2_32.dll';
```

Arguments

The only argument passed to the **inet_addr** function is the operating system address of the buffer containing the host address as a null-terminated string. You should use the the built-in function addr to get the operating system address of this buffer.

Return Values

The **inet_addr** function returns an unsigned long value containing the host address represented as a numeric value, if the call is successful. If an error occurs then the value *INADDR_NONE* is returned.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.8 The inet_ntoa function

Description

The **inet_ntoa** function converts a host address from a numeric value into a null-terminated string.

Declaration

The system include file **WinSock2.inc** contains the following declarations for the *in_addr* type, and the **inet_ntoa** function:

```
in_addr = u_long;

function inet_ntoa(inaddr : in_addr) : address;
external dll='ws2_32.dll';
```

Arguments

The only argument passed to the **inet_ntoa** function is host address, represented as a numeric value.

Return Values

The **inet_ntoa** function returns the operating system address of a buffer containing a null-terminated string representing the Internet address, if the call is successful. If the call fails zero is returned. When the **inet_ntoa** function call succeeds, the buffer returned resides in memory that is allocated by Windows Sockets. Your program The application should not make any assumptions about the way in which the memory is allocated. The data in the buffer is guaranteed to be valid until the next Windows Sockets function call within the same thread. You should copy the null-terminated string from the buffer before calling any other Windows Sockets functions.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.9 The ntohl function

Description

The **ntohl** function converts a value of type *u_long* from TCP/IP network byte order (which is big-endian) to host order.

Declaration

The system include file **WinSock2.inc** contains the following declarations for the *u_long* type, and the **ntohl** function:

```
u_long = word;

function ntohl(netlong : u_long) : u_long;
external dll='ws2_32.dll';
```

Arguments

The only argument passed to the **ntohl** function is the *u_long* value that you want to convert to host byte order.

Return Values

The **ntohl** function returns the *u_long* value in host byte order.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.10 The ntohs function

Description

The **ntohs** function converts a value of type *u_short* from TCP/IP network byte order (which is big-endian) to host order.

Declaration

The system include file **WinSock2.inc** contains the following declarations for the *u_short* type, and the **ntohs** function:

```
u_short = shortint;

function ntohs(netshort : u_short) : u_short;
external dll='ws2_32.dll';
```

Arguments

The only argument passed to the **ntohs** function is the *u_short* value that you want to convert to host byte order.

Return Values

The **ntohs** function returns the *u_short* value in host byte order.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

12.16.11 The setsockopt function

Description

The **setsockopt** function sets the current value of a socket's option.

Declaration

The system include file **WinSock2.inc** contains the following declaration for the **setsockopt** function:

```
function setsockopt(s : SOCKET; level, optname : integer; optval : address; optlen :
integer) : integer;
external dll='ws2_32.dll';
```

Arguments

The First Argument

The first argument passed to the **setsockopt** function is the socket for which you want to set the option value.

The Second Argument

The second argument passed to the **setsockopt** function is the level at which the socket option is defined. You will usually use *SOL_SOCKET* as the value of this argument.

The third argument passed to the **setsockopt** function is an integer value that identifies the socket option whose value you want to set. The socket options you are most likely to set are given below:

- SO_BROADCAST - This option is of type **BOOL**, and indicates whether you can use the socket to send broadcast messages.
- SO_DEBUG - This option is of type **BOOL**, and indicates whether debugging is enabled.
- SO_DONTLINGER - This option is of type **BOOL**, and indicates whether the *SO_LINGER* option is disabled.
- SO_DONTROUTE - This option is of type **BOOL**, and indicates whether routine is disabled.
- SO_LINGER - This opton is of type *linger* and sets the current linger options for the socket (see the closesocket function for more information about linger options).
- SO_OOBINLINE - This option is of type **BOOL**, and indicates whether OOB data will be received in the normal data stream.

Return Values

The **setsockopt** function returns a value of integer type that indicates whether the call was successful. A value of zero means that the call succeeded. A value of *SOCKET_ERROR* indicates that the called failed, and in this case you can use the WSAGetLastError function to retrieve a code that identifies the error that caused the call to fail. **NOTE:** The constant *SOCKET_ERROR* is declared in the system include file **WinSock2.inc**.

Reference Information

The authoritative source of information about the WinSock2 library is the Microsoft Developers Network (MSDN). You can access the MSDN on the Microsoft website at msdn.microsoft.com.

13.1 What are extensions?

Standard Pascal (i.e. ISO/IEC 7185) allows implementations to include extensions to Pascal. An extension is not allowed to invalidate any program that would be valid without the extension, except those programs that are invalidated only because of new keywords added by the extension.

Irie Pascal supports a number of extensions to Standard Pascal. Some of these extensions were added for compatibility with Turbo Pascal or Extended Pascal, while others are specific to Irie Pascal.

Support for extensions can be enabled and disabled using the Extensions Project Options page (see the Irie Pascal User's Manual for more information).

The Irie Pascal extensions are listed below:

- Allow relaxed declarations
- Allow constant ranges
- Allow 'otherwise'
- Allow relaxed parameter list congruity
- Allow non-numeric statement labels
- Allow underscores in identifiers
- Allow binary and hexadecimal integers
- Auto-declare Input and Output
- Allow double-quoted literals
- Enable non-standard operators
- Enable non-standard constants
- Enable non-standard types
- Enable non-standard variables
- Enable non-standard functions
- Enable non-standard procedures

13.2 Allow relaxed declarations

When the **relaxed declarations** extension is enabled, declarations can occur before the reserved word **program** which normally marks the start of the program. When this is done, the declarations are placed in the same outermost scope as the declarations for the built-in identifiers (e.g. **writeln**). As you probably know, when you declare an identifier, your declaration will override any other declarations of that identifier in outer scopes. It is recommended that you include system include files before the reserved word **program**, so that the many declarations they contain will be placed in the outermost scope, where you can easily ignore the ones you are not interested in, because if you happen to declare the same identifier, in your program, as one of the identifiers declared in the system include file, your declaration will override the declaration in the system include file.

Standard Pascal requires that all declarations and definitions of the same kind must be made together in a single group, and that the groups must appear in a specific order. The groups are listed below in the order that they must appear in a Standard Pascal program.:

- Label declaration groups - Contain label declarations.
- Constant definition groups - Contain constant definitions.
- Type definition groups - Contain type definitions.
- Variable declaration groups - Contain variable declarations.
- Sub-block declaration groups - Contain function declarations and procedure declarations.

When the **relaxed declarations** extension is enabled, there can be more than one of each kind of decaration/definition group, and groups can appear in any order except that for local declarations the sub-block declaration group must be last.

So if the **relaxed declarations** extension is enabled then the following program is valid:

```
program summing(output);
const
first = 1;
last = 100;
type
num = first..last;
var
i : num;
type
atype = array[first..last] of integer;
var
a : atype;
sum : integer;
begin
sum := 0;
for i := first to last do
begin
sum := sum + i;
a[i] := sum
end;
for i := last downto first do
writeln(i, a[i]);
end.
```

even though it has two type definition groups

`type num = first..last;`

and

`type atype = array[first..last] of integer;`

and two variable declaration groups

`var i : num;`

and

`var a : atype; sum : integer;`

In Standard Pascal (i.e. if the **relaxed declarations** extension is **not** enabled), you would have to combine these groups so you would have the following:

```
program summing(output);
const
first = 1;
last = 100;
type
num = first..last;
atype = array[first..last] of integer;
var
i : num;
a : atype;
sum : integer;
begin
sum := 0;
for i := first to last do
begin
sum := sum + i;
a[i] := sum
```

```
end;
for i := last downto first do
writeln(i, a[i]);
end.
```

13.3 Allow constant ranges

Standard Pascal requires that case constants be specified individually. When the **constant ranges** extension is enabled, you can use **constant ranges** to specify a number of consecutive case constants. To use a constant range you specify the first constant in the range, and the last constant in the range, separated by **..** as follows:

```
first..last
```

You could use the constant range

```
1..5
```

to specify the following case constants

```
1, 2, 3, 4, 5
```

For example suppose **c** is a variable of char type and you want to use a case statement to write the word **Letter** if **c** contains a letter, and the word **Digit** if **c** contains a digit, then you could specify each case constant individually as follows:

```
case c of
'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
: write('Letter');
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
: write('Digit');
end;
```

Or you could use constant ranges like the following:

```
case c of
'a'..'z', 'A'..'Z'
: write('Letter');
'0'..'9'
: write('Digit');
end;
```

Constant ranges can be used in case statement, like in the example above, and in variant record types.

13.4 Allow 'otherwise'

When the **allow otherwise** extension is enabled, Irie Pascal allows you to use the reserved words **otherwise** or **else**, in case statements and variant record types to specify *all case constants that haven't already been specified*.

For example in the following variant record

```
type
character = record
case c : char of
```

```
'a'..'z', 'A'..'Z'
: (vowel : Boolean);
'0'..'9'
: (value : integer);
otherwise
();
end;
```

at the point that **otherwise** is used, the following case constants have already been specified

```
'a'..'z', 'A'..'Z', and '0'..'9'
```

so **otherwise** specifies the remaining case constants, which in this case are all the other values of the char type.

13.5 Allow relaxed parameter list congruity

As an extension to Standard Pascal, Irie Pascal relaxes the rules when deciding whether two formal parameter lists are *congruent*. **NOTE:**This extension is the only one that can not be disabled.

A rarely used feature of the Pascal language is the ability to pass functions and procedures to other functions and procedures. Functions are passed using functional parameters, and procedures are passed using procedural parameters. Functional and procedural parameters come in two forms, actual and formal, just like the other kinds of parameters (value and variable parameters). Actual functional parameters specify the functions being passed, and formal functional parameters are used to reference the functions being passed, inside the receiving functions or procedures. In a similar way, actual procedural parameters specify the procedures being passed, and formal procedural paramters are used to reference the procedures being passed, inside the receiving functions or procedures.

The formal parameter list of the function specified by an actual functional parameter must be *congruent* with the formal parameter list of the corresponding formal functional parameter. The formal parameter list of the procedure specified by an actual procedural parameter must be *congruent* with the formal parameter list of the corresponding formal procedural parameter.

Standard Pascal takes into account the formal parameter sections when deciding whether two formal parameter lists are *congruent*, and requires that they must have the same number of formal parameter sections (amoung other requirements). Irie Pascal ignores formal parameter sections and compares individual parameters.

So for example, Irie Pascal considers the following program to be valid, but Standard Pascal does not.

```
program functional(output);

function Operation(function op(a, b : integer) : integer; i, j : integer) : integer;
begin
Operation := op(i, j);
end;

function Add(c : integer; d : integer) : integer;
begin
Add := c + d;
end;

begin
writeln(Operation(Add, 3, 1));
end.
```

The formal parameter list of the **function Add** contains two formal parameter sections:

```
c : integer
```

and

```
d : integer
```

while the formal parameter list of the **formal functional parameter op** contains one formal parameter section:

```
a, b : integer
```

and so Standard Pascal does **not** consider the two formal parameter lists to be *congruent*, and therefore the following

```
Operation(Add, 3, 1)
```

is invalid. Irie Pascal compares corresponding formal parameters idividually, so

```
c : integer
```

is compared with the first parameter in

```
a, b : integer
```

and they are both value parameters of integer type, and

```
d : integer
```

is compared with the second parameter in

```
a, b : integer
```

and again they are both value parameters of integer type, so Irie Pascal considers the two formal parameter lists to be *congruent*, and therefore the following

```
Operation(Add, 3, 1)
```

is valid.

13.6 Allow non-numeric statement labels

When the **allow non-numeric statement labels** extension is enabled, Irie Pascal supports statement labels that look like identifiers (i.e. can contain letters and underscore characters).

For example, in the following program, **loop** is used as a statement label.

```
program name(output);
label loop;
var
i : integer;
begin
i := 1;
loop: writeln(i);
i := i + 1;
if i <= 20 then goto loop;
end.
```

13.7 Allow underscores in identifiers

When the **allow underscores in identifiers** extension is enabled, Irie Pascal supports identifiers which contain and/or start with underscore characters (i.e. '_'). So for example the following would be valid identifiers:

```
_name
last_name
_all_names_
```

13.8 Allow binary and hexadecimal integers

When the **allow binary and hexadecimal integers** extension is enabled, Irie Pascal supports integers specified using binary notation and hexadecimal notation. In Standard Pascal, or when this extension is disabled, integers can only be specified using decimal notation.

13.9 Auto-declare Input and Output

Standard (ISO/IEC 7185) Pascal specifies that whenever the required identifiers input and output are referenced in a program, they must appear as program parameters. When the **auto-declare input and output** extension is enabled, Irie Pascal does **not** require that input and output appear as program parameters.

For example the following program:

```
program hello;
begin
writeln('Hello!!')
end.
```

is equivalent to

```
program hello(output);
begin
writeln('Hello!!')
end.
```

The first program is not valid in Standard Pascal because the required identifier output is indirectly referenced by the use of the writeln procedure without specifying a file variable, but output does not appear as a program parameter.

13.10 Allow double-quoted literals

When the **allow double-quoted literals** extension is enabled, Irie Pascal supports character and string literals enclosed with double quotation marks.

For example you could use

```
"Hello world"
```

instead of

```
'Hello world'
```

Double-quoted literals can be especially useful if you want to create literals that contain single quotes since you don't have to use two single quotes to represent one single quote.

For example you could create a string literal containing

```
Don't go away
```

with the following:

```
"Don't go away"
```

instead of

```
'Don''t do away'
```

13.11 Enable non-standard operators

When the **enable non-standard operators** extension is enabled, Irie Pascal supports the use of non-standard operators, and the use of standard operators in non-standard ways.

The non-standard operators are listed below:

- and_then
- or_else
- shl (bit shift left)
- shr (bit shift right)
- xor (boolean)
- xor (bitwise)

The standard operators that can be used in non-standard ways are listed below:

- not (bitwise)
- and (bitwise)
- or (bitwise)
- + (string concatenation)

13.12 Enable non-standard constants

When the **enable non-standard constants** extension is enabled, Irie Pascal supports the use of the following non-standard constants:

- appendmode
- dir_bit
- feature_calldll
- feature_closedir
- feature_clrscr
- feature_delay
- feature_gotoxy
- feature_idispatch
- feature_intr
- feature_keypressed
- feature_mysql
- feature_odbc

- feature_opendir
- feature_popen
- feature_readdir
- feature_readkey
- feature_rewinddir
- feature_textbackground
- feature_textcolor
- feature_wherex
- feature_wherey
- grp_r
- grp_w
- grp_x
- maxbyte
- maxchar
- maxword
- oth_r
- oth_w
- oth_x
- platform_dos
- platform_error
- platform_fbsd
- platform_linux
- platform_os2
- platform_solaris
- platform_solaris_sparc
- platform_win32
- readmode
- rsdefault
- rsdynamic
- rsforward
- rskeyset
- rsstatic
- sql_tc_all
- sql_tc_ddl_commit
- sql_tc_ddl_ignore
- sql_tc_dml
- sql_tc_none
- usr_r
- usr_w
- usr_x
- writemode

13.13 Enable non-standard types

When the **enable non-standard types** extension is enabled, Irie Pascal supports the use of the following non-standard types:

- address type
- binary type
- byte type
- connection type
- dir type
- double type
- error type

- filename type
- list types
- recordset type
- registers type
- regtype type
- shortint type
- shortword type
- single type
- word type

13.14 Enable non-standard variables

When the **enable non-standard variables** extension is enabled, Irie Pascal supports the use of the following non-standard variables:

- errors
- exitcode
- null

13.15 Enable non-standard functions

When the **enable non-standard functions** extension is enabled, Irie Pascal supports the use of the following non-standard functions:

- The addr function
- The concat function
- The copy function
- The copyword function
- The cosh function
- The countwords function
- The createobject function
- The dirsep function
- The fexpand function
- The filematch function
- The filepos function
- The filesize function
- The frac function
- The getenv function
- The getlasterror function
- The hex function
- The int function
- The ioresult function
- The isalpha function
- The isalphanum function
- The isdigit function
- The islower function
- The isnull function
- The isprint function
- The isspace function
- The isupper function
- The isxdigit function
- The keypressed function
- The length function

- The log function
- The locase function
- The lowercase function
- The paramcount function
- The paramstr function
- The pi function
- The platform function
- The pos function
- The ptr function
- The random function
- The readkey function
- The reverse function
- The sinh function
- The sizeof function
- The stopserverviceevent function
- The supported function
- The swap function
- The system function
- The tanh function
- The trim function
- The unixplatform function
- The upcase function
- The uppercase function
- The urldecode function
- The version function
- The wait function
- The wherex function
- The wherey function

13.16 Enable non-standard procedures

When the **enable non-standard procedures** extension is enabled, Irie Pascal supports the use of the non-standard procedures, and the use of standard procedures in non-standard ways:

The non-standard procedures are listed below:

- The append procedure
- The assert procedure
- The assign procedure
- The chdir procedure
- The close procedure
- The closedir procedure
- The clrscr procedure
- The crc32 procedure
- The dec procedure
- The delay procedure
- The delete procedure
- The erase procedure
- The exec procedure
- The exit procedure
- The fill procedure
- The flush procedure
- The fsplit procedure
- The getdate procedure

- The getfiledate procedure
- The getfilemode procedure
- The getfiletime procedure
- The gettime procedure
- The gotoxy procedure
- The halt procedure
- The inc procedure
- The insert procedure
- The intr procedure
- The mkdir procedure
- The move procedure
- The msdos procedure
- The open procedure
- The opendir procedure
- The popen procedure
- The randomize procedure
- The rawread procedure
- The rawwrite procedure
- The readdir procedure
- The rename procedure
- The rewinddir procedure
- The rmdir procedure
- The seek procedure
- The setfiledate procedure
- The setfiletime procedure
- The sleep procedure
- The str procedure
- The textbackground procedure
- The textcolor procedure
- The traperrors procedure
- The val procedure

The standard procedures that can be used in non-standard ways are listed below:

- The reset procedure
- The rewrite procedure

14.1 Deviations from ISO/IEC 7185

Irie Pascal complies with the requirements of level 0 of Standard Pascal (i.e. ISO/IEC 7185), with the following exceptions:

- end-of-line char
- Termination of all lines in text files
- new(p, c1..cN)
- dispose(q, k1..kM)
- Carriage returns

ISO/IEC 7185 is the standard for the Pascal porgramming language published by the Internation Organization for Standardization.

14.2 end-of-line char

Standard Pascal requires that end-of-line characters read from text files be converted to spaces. For example given

```
read(f, c)
```

where **f** is a text file at end-of-line and **c** is a character variable then according to Standard Pascal, after the read above, **c** should contain a space character rather than an end-of-line character. Irie Pascal does not convert end-of-line characters read from text files, so in the example above **c** will contain an end-of-line character (i.e. a linefeed character, chr(10)).

14.3 Termination of all lines in text files

Standard Pascal requires that all lines in text files (except possibly the line that is currently being written) should be terminated by an end-of-line. In effect this means that after a text file is closed the last line must be terminated by an end-of-line. Irie Pascal does not automatically terminate the last line in a text file.

14.4 new(p, c1..cN)

Standard Pascal requires support for a form of the required procedure new as follows:

```
new(p, c1..cN)
```

where **p** is a pointer to a variant record and **c1** thru **cN** are case constants which correspond to the variants in **p^**. Irie Pascal does not currently support this form of new.

14.5 dispose(q, k1..kM)

Standard Pascal requires support for a form of the required procedure dispose as follows:

```
dispose(q, k1..kM)
```

where **q** is a pointer to a variant record and **k1** thru **kM** are case constants which correspond to the variants in **q^**. Irie Pascal does not currently support this form of dispose.

14.6 dispose(pointer value)

In Standard Pascal, the parameter passed to the dispose procedure is an expression of pointer type. However in Irie Pascal, the parameter passed to the dispose procedure is a reference to a pointer variable.

14.7 Carriage returns

Standard Pascal requires that all characters written to a file should appear when the file is read (the exception to this is that implementations are allowed to designate a set of characters as *prohibited from text files* and treat these characters specially). The Windows edition of Irie Pascal does not comply fully with this requirement, since with this edition carriage-return characters do not appear when individual characters are read from text files. This treatment of carriage-return characters is intended to convert carriage-return/line-feed pairs (that are used as end-of-line markers in some text files) into a single end-of-line character (i.e. a line-feed chr(10)).

15.1 Implementation limits

Standard Pascal allows implementations to impose limits on the size of a program and its data. Irie Pascal imposes various limits on the size and complexity of programs and their data, that it will process. These limits are listed below:

- Length of source lines
- Number and nesting of statement sequences
- Nesting of functions/procedures
- Range of variant selector's type & Number of variants
- Size of integer constants
- Size of real constants
- Size of code and data
- Number of set elements
- Nesting of statements
- Nesting of include directives
- Stack size
- Number of Source Lines

15.2 Length of source lines

The maximum length of a source line, that the Irie Pascal compiler will process, is 400 characters. If you exceed this limit the compiler will issue an error message.

15.3 Number and nesting of statement sequences

Statement sequences are the statements that occur between **begin** and **end** and between **repeat** and **until**. Statement sequences are nested when one statement sequence is contained in another.

The maximum number of statement sequences in a function or procedure is 2,147,486,647. If you exceed this limit the Irie Pascal compiler issues a fatal error message. Statement sequences can be nested to a maximum depth of 256. If you exceed this limit the compiler issues a fatal error message.

15.4 Nesting of functions/procedures

Pascal supports nested functions and procedures (i.e. functions and procedures can contain other functions and procedures). Irie Pascal supports nesting functions and procedures to a maximum depth of 256. **NOTE:** This does **not** mean that the total number of functions and procedures in a program is 256. This limits applies only to the depth of nesting. If you exceed this limit the compiler issues a fatal error message.

15.5 Range of variant selector's type & Number of variants

Irie Pascal limits the maximum range of the tag-type of a variant record's variant selector to 1024 (i.e. the maximum number of values specified by the tag-type is 1024). This also indirectly limits the number of variants in a variant record's variant-part to 1024. If you exceed this limit the compiler issues a warning message and the following checks are not performed on the variant record which exceeded the limit(s).

- No case-constant should appears more than once.
- Each value specified by the tag-type should appear as a case-constant.
- Only active variants in variant records are accessed.

15.6 Size of integer constants

The maximum size of an integer constant is **4,294,967,295**. If you exceed this limit the compiler will issue an error message.

15.7 Size of real constants

The maximum size of a real constant is approximately **1e308**. If you exceed this limit the compiler will issue an error message.

15.8 Size of code and data

The maximum size of the code in a program is **4GB**. The maximum size of the data in a program is **4GB**. If you exceed these limits the compiler will issue an error message.

15.9 Number of set elements

The maximum number of set elements that can be present in a set value is 256. **NOTE:** This limit does not apply to the range of the values of set elements in set values. For example in the following program:

```
program testset;
var
setvar : set of 1..2000;
begin
setvar := [1, 1000, 2000];
writeln(1 in setvar);
writeln(1000 in setvar);
writeln(2000 in setvar);
end.
```

Although the range of values in the set constructor is 2,000, the number of set elements actually present in the set constructor is three, so this program runs without a problem.

15.10 Nesting of statements

Irie Pascal places two limits on the maximum depth to which statements can be nested.

The first limit is a general limit and limits the maximum depth to which any statement can be nested inside any other statement. This limit is 256. **NOTE:** This does **not** mean that the maximum number of statements in a program is 256. This limits the depth of nesting (i.e. the depth to which statements contain other statements which in turn contain other statements). If you exceed this limit the compiler issues a fatal error message.

The second limit is more specific and limits the maximum depth to which the following kinds of statements can be directly or indirectly nested inside each other:

- Case statements

- For statements
- With statements

This second limit is 20. If you exceed this limit the compiler issues a error message and continues.

15.11 Nesting of include directives

Irie Pascal limits the depth to which files can be included into each other. This limit is 30. **NOTE:** Files can be included into each other using the include compiler directive. The syntax for the include compiler directive is

(*$I name*)

or

{$I name}

where **name** is the name of the file to include into the current file.

15.12 Stack size

The default stack size is 64K but you can adjust this to any value from 1K to 1024K (1MB) using the Code Generation Page of the Project Options dialog. If you exceed this limit the compiler will issue an error message.

15.13 Number of Source Lines

Irie Pascal limits the maximum number of source lines that can be compiled to 4,294,967,295. If you exceed this limit the compiler will issue a fatal error message.

16.1 Implementation-defined features

The Pascal Standard (ISO/IEC 7185) says that some features of the Pascal language are implementation-defined. ISO/IEC 7185 further says that each implementation (called a processor by ISO/IEC 7185) must provide a definition of all implementation-defined features.

Below are links to definitions of how Irie Pascal handles implementation-defined features. Each definition has the following format:

First the official description of the feature is taken from ISO/IEC 7185 and displayed in red. This is followed by a simplified description, if the official description is unclear or requires the reader to have a copy of ISO/IEC 7185. Finally a statement describing how Irie Pascal handles the feature is at the end.

- String-elements
- Provision of tokens and delimiting characters
- Size and precision of real values
- Character set
- Ordinal values of characters
- Characters prohibited from text files
- When I/O is performed

16.2 String-elements

6.1.7 There shall be an implementation-defined one-to-one correspondence between the set of alternatives from which string-elements are drawn and a subset of the values of the required char-type.

In other words, the characters that form string literals and character literals, is implementation-defined.

Irie Pascal allows string literals and character literals to be formed from any printable character.

16.3 Provision of tokens and delimiting characters

6.1.9 Provision of the reference tokens ^, [, and], of the alternative token @, and of the delimiting characters { and }, is implementation-defined.

Irie Pascal provides the reference tokens, the alternative token, and the delimiting characters.

16.4 Size and precision of real values

6.4.2.2 b) The values of the real-type shall be an implementation-defined subset of the real numbers denoted by signed-real.

The real numbers described by the ISO/IEC 7185 have infinite range and infinite precision. Implementations are allowed to limit the maximum range and precision of real numbers.

Irie Pascal uses 64 bit reals stored as specified by the following standard: "IEEE standard for binary floating-point arithmetic (ANSI/IEC std 754-1985)".

16.5 Character set

6.4.2.2 d) The values of char-type shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representation.

Values of char type in Irie Pascal are the enumeration of all characters, printable and non-printable.

16.6 Ordinal values of characters

6.4.2.2 d) The ordinal numbers of each value of char-type are implementation-defined.

Irie Pascal uses the position of each value of char type, in the current character set, as it's ordinal number. So for example since the space character is at position 32 then

```
ord(' ') = 32
```

16.7 Characters prohibited from text files

6.4.3.5 There shall be an implementation-defined subset of the set of char-type values, designated 'characters prohibited from text files'.

This appears to be a loophole provided by ISO/IEC 7185 to allow implementations to get around the requirements specified for text files, especially the requirement that all characters written to a text file appear when the text file is read in.

Irie Pascal designates one character as **prohibited from text files**. This character is chr(26), also known as the End-Of-File (EOF) character. NOTE: The fact that the EOF character is **prohibited from text files** doesn't mean that you can't write them to text files, but if you do then the text file will be terminated at that point (i.e. the EOF and the characters written after the EOF will not appear when the text file is read back in). Since this technically violates the standard, EOF's are prohibited from text files.

16.8 When I/O is performed

6.6.5.2 The point at which the file operations "rewrite", "put", "reset", and "get" are actually performed is implementation-defined.

Irie Pascal implements **lazy I/O** (i.e. input operations are performed at the latest opportunity and output operations are performed at the earliest opportunity) in order to facilitate interactive terminal input and output. The file operations are performed as specified below:

- rewrite - the file is opened but the file buffer is empty.
- put - the contents of the file buffer are appended to the file.
- reset - the file is opened, but the file buffer is not filled until it is accessed.
- get - If the file buffer is full then the file buffer is emptied to make room for the next file value (NOTE: In this case no I/O is actually performed). If the file buffer is empty then the next data value is skipped (i.e. read and ignored). The file buffer remains empty.

16.9 Value of "maxint"

6.7.2.2 The required constant-identifier maxint shall denote an implementation-defined value of integer-type.

Irie Pascal uses 2,147,483,647 as the value of maxint.

16.10 Accuracy of real operations and functions

6.7.2.2 The accuracy of the approximation of the result of the real operations and functions to the mathematical result is implementation-defined.

Irie Pascal performs real operations as specified in "IEEE standard for binary floating-point arithmetic (ANSI/IEC std 754-1985)", and the final results are stored in 64 bit reals.

16.11 Default value of TotalWidth for integer-type

6.9.3.1 The default value of TotalWidth for integer-type is implementation-defined.

In other words, when writing integer values to text files, the default width of the output field is implementation-defined.

Irie Pascal uses 8 as the default value of TotalWidth for integer type.

So for example

```
write('one=',1)
```

will cause

```
one=       1
```

to be written (i.e. the integer value 1 is written as seven spaces and a 1).

16.12 Default value of TotalWidth for real-type

6.9.3.1 The default value of TotalWidth for real-type is implementation-defined.

In other words, when writing real values to text files, the default width of the output field is implementation-defined.

Irie Pascal uses 9 as the default value of TotalWidth for real type.

16.13 Default value of TotalWidth for boolean-type

6.9.3.1 The default value of TotalWidth for boolean-type is implementation-defined.

In other words, when writing boolean values to text files, the default width of the output field is implementation-defined.

Irie Pascal uses 8 as the default value of TotalWidth for boolean type.

16.14 Number of digits written in exponents

6.9.3.4.1 ExpDigits is an implementation-defined value representing the number of digit-characters written in an exponent.

The value of ExpDigits is three (3) for the Windows edition of Irie Pascal, and two (2) for the other editions. So for example

```
write('one=', 1.0)
```

will write

```
one= 1.0E+000
```

or

```
one= 1.0E+00
```

depending on the edition of Irie Pascal being used.

16.15 Use of "e" or "E" as exponent character on output

6.9.3.4.1 The value of the exponent character ('e' or 'E') used on output of values of real-type is implementation-defined.

Irie Pascal uses **E** on output of values of real type.

16.16 Case of characters used on output of boolean values

6.9.3.5 The case of each character of 'True' and 'False' used on output of values of boolean-type is implementation-defined.

Irie Pascal uses **true** and **false** on output of values of boolean type.

16.17 Effect of built-in procedure "page"

6.9.5 The effect of the procedure "page" when applied to a text file which is in generation mode is implementation-defined.

The procedure page causes a form feed character (i.e. chr(12)) to be written to text files to which it is applied.

16.18 Binding of file-type program-parameters

6.10 The binding of a file-type program-parameter is implementation-defined.

See what are program parameters for a description of how file type program parameters are handled.

16.19 Effect of "reset" and "rewrite" on "input" and "output"

6.10 The effect of "reset" and "rewrite" on the standard files "input" and "output" is implementation-defined.

reset and rewrite have no effect on the standard files input and output.

17.1 Implementation-dependent features

The Pascal Standard (ISO/IEC 7185) does not completely define all features of the language. Some features are implementation-dependent. Below are links to the definitions of how Irie Pascal treats all implementation-dependent features. Each definition has the following format:

First the official description of the feature is taken from ISO/IEC 7185 and displayed in red. This is followed by a simplified description, if the official description is unclear or requires the reader to have a copy of ISO/IEC 7185. Finally a statement describing how Irie Pascal treats the feature is at the end.

17.2 Effect of writing "prohibited" characters

6.4.3.5 For any text file t, the effect of attributing to a component of either t.L or t.R a member of the characters prohibited from text files is implementation-dependent.

In other words the effect of writing, any of the characters that implementations are allowed to **prohibit** from text files, to a text file is implementation-dependent.

Irie Pascal **prohibits** one character from text files, the character is chr(26) the end-of-file character. Writing the end-of-file character to a text file causes the file to be terminated (i.e. the EOF character and any characters written after the EOF character will not appear when the file is read).

17.3 Order of evaluation of index-expressions

6.5.3.2 The order of evaluation of the index-expressions of an indexed-variable is implementation-dependent.

Irie Pascal evaluates index expressions of indexed variables from left to right. For example given

```
var   x[1..10, 1..20] of integer;
```

and later on

```
x[i, j] := 20;
```

i will get evaluated before **j**.

17.4 Order of evaluation of expressions of a member-designator

6.7.1 The order of evaluation of expressions of a member-designator is implementation-dependent.

Irie Pascal evaluates expressions of member-designators from left to right. For example given

```
const low = 'a'; high = 'z';
var letter : set of char;
```

and later on

```
letter := [low..high];
```

low will be evaluated before **high**.

17.5 Order of evaluation of member-designators

6.7.1 The order of evaluation of the member-designators of a set-constructor is implementation-dependent.

Irie Pascal evaluates member-designators of set-constructors from left to right. For example given

```
var yes : set of char;
```

and later on

```
yes := ['y', 'Y']'
```

'y' will be evaluated before **'Y'**.

17.6 Order of evaluation of operands of dyadic operators

6.7.2.1 The order of evaluation of the operands of a dyadic operator is implementation-dependent.

In other words, for operators with two operands (left and right) the order in which the operands are evaluated is implementation-dependent. The operands can be evaluation left-to-right, right-to-left, in parallel, or they may not both be evaluated.

Irie Pascal always evaluates the operands of the following operators from left-to-right:

- Addition (+)
- Subtraction (-)
- multiplication (*)
- Real division (/)
- Integer division (div)
- Modulus (mod)
- Is equal (=)
- Is not equal (<>)
- Is less than (<)
- Is less than or equal (<=)
- Is greater than (>)
- Is greater than or equal (>=)
- xor (boolean)
- String concatenation (+)
- Set union (+)
- Set difference (-)
- Set intersection (*)
- Set inclusion (in)
- and (bitwise)
- or (bitwise)
- xor (bitwise)
- Bit shift left (shl)
- Bit shift right (shr)

Irie Pascal always evaluates the operands of the following operators using short-circuit evaluation:

- and_then (boolean)
- or_else (boolean)

Irie Pascal evaluates the operands of the following operators, using short-circuit evaluation if short-circuit evaluation is enabled, or left-to-right if short-circuit evaluation is disabled:

- and (boolean)
- or (boolean)

17.7 Order of evaluation of actual-parameters for function calls

6.7.3 The order of evaluation, accessing, and binding of the actual-parameters of a function-designator is implementation-dependent.

In other words, the order in which the actual parameters are evaluated and passed to the formal parameters during a functions call is implementation-dependent.

Irie Pascal evaluates and passes actual parameters from left to right.

17.8 Order of evaluating components of assignment statements

6.8.2.2 The order of accessing the variable and evaluating the expression of an assignment-statement is implementation-dependent.

Programs generated by Irie Pacal calculate the address of the variable before the expression is evaluated.

17.9 Order of evaluation of actual-parameters for procedure calls

6.7.3 The order of evaluation, accessing, and binding of the actual-parameters of a procedure-statement is implementation-dependent.

In other words, the order in which the actual parameters are evaluated and passed to the formal parameters during a procedure call is implementation-dependent.

Irie Pascal evaluates and passes actual-parameters from left to right.

17.10 Effect of reading a text file to which "page" has been applied

6.9.5 The effect of inspecting a text file to which the page procedure was applied during generation is implementation-dependent.

Applying page to a text file during generation causes a form feed character (i.e. chr(12)) to be written to the file. During inspection (i.e. when the text file is read) the handling of the form feed character depends on the type of data the program is trying to read. If the program is trying to read in integer or real data then the form feed character is treated like a space character and is therefore skipped. If the program is trying to read in character or string data then the form feed character is read in like any other character.

17.11 Binding of non-file program-parameters

6.10 The binding of the non-file variables denoted by the program-parameters to entities external to the program is implementation-dependent.

See what are program parameters for a description of the binding of program parameters to entities external to the program.

18.1 What are errors?

ISO/IEC 7185 defines an error as follows:

A violation by a program of the requirements of this International Standard that a processor is permitted to leave undetected.

ISO/IEC 7185 then goes on to add the following two notes:

1. If it is possible to construct a program in which the violation or non-violation of this International Standard requires knowledge of the data read by the program or the implementation definition of implementation-defined features, then violation of that requirement is classified as an error. Processors may report on such violations of the requirement without such knowledge, but there are always remain some cases that require execution, simulated execution, or proof procedures with the required knowledge. Requirements that can be verified without such knowledge are not classified as errors.
2. Processors should attempt the detection of as many errors as possible, and to as complete a degree as possible. Permission to omit detection is provided for implementations in which the detection would be an excessive burden.

Below are links to documentation of how Irie Pascal handles all errors. Most of the time the documentation consists only of a statement about whether or not the error is reported. The description of each error is taken from ISO/IEC 7185.

- Array index out of bounds
- Accessing inactive variant
- Dereferencing nil pointers
- Dereferencing undefined pointers
- Dangling pointers
- Altering file-variables
- Using out-of-range value parameters
- Using out-of-range set value parameters
- Output to file not open for writing
- Output to undefined file
- Writing to middle of file
- Using Put on undefined buffer-variables
- Resetting undefined files
- Input from file not open for reading
- Input from undefined file
- Reading past end-of-file
- Reading out of range values
- Writing out of range values
- new(p, c1..cN) constraints violated
- Incompatible use of dispose(p)
- Incompatible use of dispose(p, k1..kM)
- dispose(p, k1..kM) constraint violations
- disposing nil pointer
- disposing undefined pointer
- new(p, c1..cM) constraint violated
- Invalid use of pack (1)
- Invalid use of pack (2)

18.2 Array index out of bounds

Official Description

6.5.3.2 For an indexed-variable closest-containing a single index-expression, it is an error if the value of the index-expression is not assignment-compatible with the index-type of the array-type.

Simplified Description

In other words, it is error to access an array component that does not exist. For example given:

```
var a : array[1..20] of char;
```

then

```
a[0]
```

is an error since the index-expression (i.e. **0**) is not assignment-compatible with the index-type (i.e. **1..10**) of the array type. NOTE: There is no 0th component of the array **a**.

Error Handling

This error is reported if range checking is enabled.

18.3 Accessing inactive variant

Official Description

6.5.3.3 It is an error unless a variant is active for the entirety of each reference and access to each component of the variant.

Error Handling

This error is reported if the following two conditions are met:

1. Range checking is enabled.
2. The number of values defined by the variant selector's type is less than or equal to 1024.

18.4 Dereferencing nil pointers

Official Description

6.5.4 It is an error if the pointer-variable of an identified-variable denotes a nil-value.

Simplified Description

In other words, it is an error to dereference a pointer variable which is equal to nil (i.e. p^ is an error if p = nil).

Error Handling

This error is always reported.

18.5 Dereferencing undefined pointers

Official Description

6.5.4 It is an error if the pointer-variable of an identified-variable is undefined.

Simplified Description

In other words, it is an error to dereference a pointer variable which is undefined (i.e. p^ is an error if p is undefined).

Error Handling

This error is always reported.

18.6 Dangling pointers

Official Description

6.5.4 It is an error to remove from its pointer-type the identifying-value of an identified-variable when a reference to the identified-variable exists.

Simplified Description

In other words, it is an error to use dispose to free the memory allocated for a pointer if other pointers to the same memory exist.

Error Handling

This error is never reported.

18.7 Altering file-variables

Official Description

6.5.5 It is an error to alter the value of a file-variable **f** when a reference to the buffer-variable **f^** exists.

Simplified Description

According to ISO/IEC 7185 there are four ways of establishing a reference to a variable.

1. Passing the variable by reference establishes a reference to the variable for the life of the function or procedure call.
2. Using an assignment statement to assign a value to the variable, establishes a reference to the variable starting either before or after the expression on the right-hand side of the assignment statement is evaluated, and lasting until the value has been assigned to the variable.
3. Using the variable in a with statement establishes a reference to the variable that exists during the execution of the statement in the with statement.
4. Establishing a reference to a component of the variable, also establishes a reference to the variable as a whole.

So for example, it would be an error if you pass a file buffer by reference to a procedure and that procedure performs I/O on the file associated with the file buffer.

Error Handling

This error is never reported.

18.8 Using out-of-range value parameters

Official Description

6.6.3.2 It is an error if the value of each corresponding actual value parameter is not assignment-compatible with the type possessed by the formal-parameter.

Simplified Description

In other words, when you pass values into formal parameters, during a function or procedure call, the values must be assignment compatible with the type of the formal parameters into which they are passed. If you think about it, this makes sense since passing a value into a formal parameter is very much like assigning the value to the formal parameter.

Error Handling

This error is reported if range checking is enabled.

18.9 Using out-of-range set value parameters

Official Description

6.6.3.2 For a value parameter, it is an error if the actual-parameter is an expression of a set-type whose value is not assignment compatible with the type possessed by the formal-parameter.

Simplified Description

In other words, when you pass set values into formal parameters, during a function or procedure call, the values must be assignment compatible with the type of the formal parameters into which they are passed. If you think about it, this makes sense since passing a value into a formal parameter is very much like assigning the value to the formal parameter.

Error Handling

This error is reported if range checking is enabled.

18.10 Output to file not open for writing

Official Description

6.6.5.2 It is an error if the file mode is not **generation** immediately prior to any use of put, write, writeln or page.

Simplified Description

In other words, it is an error to perform output on a file which is not open for writing (i.e. the file is either not open or open for reading).

Error Handling

This error is reported if I/O checking is enabled.

18.11 Output to undefined file

Official Description

6.6.5.2 It is an error if the file is undefined immediately prior to any use of put, write, writeln or page.

Simplified Description

In other words, it is an error to perform output on a file which does not exist.

Error Handling

This error is reported if I/O checking is enabled.

18.12 Writing to middle of file

Official Description

6.6.5.2 It is an error if end-of-file is not true immediately prior to any use of put, write, writeln or page.

Simplified Description

In other words, all file output must take place at the end of files.

Error Handling

Irie Pascal supports output in the middle of a file using the seek procedure, and so does not consider this an error and therefore does not report it.

18.13 Using Put on undefined buffer-variables

Official Description

6.6.5.2 It is an error if the buffer-variable is undefined immediately prior to any use of put.

Simplified Description

In other words, it is an error to use the put procedure to write the contents of a file variable's buffer variable, if its contents are undefined.

Error Handling

This error is reported if I/O checking is enabled.

18.14 Resetting undefined files

Official Description

6.6.5.2 It is an error if the file is undefined immediately prior to any use of reset.

Simplified Description

In other words, it is an error to use the reset procedure (which opens files for reading) to open a file, if the file does not exist.

Error Handling

This error is reported if I/O checking is enabled.

18.15 Input from file not open for reading

Official Description

6.6.5.2 It is an error if the file mode is not **inspection** immediately prior to any use of get or read.

Simplified Description

In other words, it is an error to perform input from a file which is not open for reading (i.e. the file is either not open or open for writing).

Error Handling

This error is reported if I/O checking is enabled.

18.16 Input from undefined file

Official Description

6.6.5.2 It is an error if the file is undefined immediately prior to any use of get or read.

Simplified Description

In other words, it is an error to perform input on a file which does not exist.

Error Handling

This error is reported if I/O checking is enabled.

18.17 Reading past end-of-file

Official Description

6.6.5.2 It is an error if end-of-file is true immediately prior to any use of get or read.

Simplified Description

In other words, it is an error to attempt to read past the end of a file.

Error Handling

This error is reported if I/O checking is enabled.

18.18 Reading out of range values

Official Description

6.6.5.2 For read, it is an error if the value possessed by the buffer-variable is not assignment-compatible with the variable-access.

Simplified Description

In other words, it is an error to read a value into a variable, if the value is not assignment compatible with the variable.

Error Handling

This error is reported if range checking is enabled.

18.19 Writing out of range values

Official Description

6.6.5.2 For write, it is an error if the value possessed by the expression is not assignment-compatible with the buffer-variable.

Simplified Description

In other words, it is an error to write a value to a file, if the value is not assignment compatible with the file type's component type. Or to put it another way, it is an error to write a value to a file, if the value is not a valid value for that file. For example given:

```
type
positive = 0..maxint;
var
f : file of positive;
```

then

```
write(f, -1);
```

is an error since -1 is not assignment compatible with **0..maxint**.

Error Handling

This error is reported if range checking is enabled.

18.20 new(p, c1..cN) constraints violated

Official Description

6.6.5.3 For **new(p, c1..cN)**, it is an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

Simplified Description

In other words, if you use this form of the new procedure and specify case constants **c1** thru **cN** then it is an error if a variant other than those specified by **c1** thru **cN** become active.

Error Handling

Irie Pascal does not support this form of the new procedure so this error can not occur.

18.21 Incompatible use of dispose(p)

Official Description

6.6.5.3 For **dispose(p)**, it is an error if the identifying-value had been created using the form **new(p, c1..cN)**.

Error Handling

Irie Pascal does not support this form of the new procedure so this error can not occur.

18.22 Incompatible use of dispose(p, k1..kM)

Official Description

6.6.5.3 For **dispose(p, k1..kM)**, it is an error unless the variable had been created using the form **new(p, c1..cN)** and **m** is equal to **n**.

Error Handling

Irie Pascal does not support this form of the dispose procedure so this error can not occur.

18.23 dispose(p, k1..kM) constraint violations

Official Description

6.6.5.3 For **dispose(p, k1..kM)**, it is an error if the variants identified by the pointer value of **p** are different from those specified by the case-constants **k1..kM**.

Error Handling

Irie Pascal does not support this form of the dispose procedure so this error can not occur.

18.24 disposing nil pointer

Official Description

6.6.5.3 For dispose, it is an error if the pointer variable has a nil-value.

Error Handling

This error is always reported.

18.25 disposing undefined pointer

Official Description

6.6.5.3 For dispose, it is an error if the pointer variable has an undefined value.

Error Handling

This error is always reported.

18.26 new(p, c1..cM) constraint violated

Official Description

6.6.5.3 It is an error if a variable created using the second form of **new** is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual parameter.

Error Handling

Irie Pascal does not support this form of the new procedure so this error can not occur.

18.27 Invalid use of pack (1)

Official Description

6.6.5.4 For pack, it is an error if the parameter of ordinal-type is not assignment compatible with the index-type of the unpacked array parameter.

Simplified Description

In other words, the parameter used to specify the first element of the unpacked array to transfer to the packed array must be assignment compatible with the index type of the unpacked array. Or to put it another way the first element of the unpacked array to transfer to the packed array must exist. For example given:

```
var
unpacked_value : array[1..10] of integer;
packed_value : packed array[21..30] of integer;
```

then

```
pack(unpacked_value, 0, packed_value);
```

is an error since **0** is not assignment compatible with **1..10**. Or to put it another way element 0 of the unpacked array does not exist.

Error Handling

This error is always reported.

18.28 Invalid use of pack (2)

Official Description

6.6.5.4 For pack, it is an error if any of the components of the unpacked array are both undefined and accessed.

Simplified Description

In other words, it is an error if the pack procedure attempts to transfer undefined array elements from the unpacked array.

Error Handling

This error is never reported.

18.29 Invalid use of pack (3)

Official Description

6.6.5.4 For pack, it is an error if the index-type of the unpacked array is exceeded.

Simplified Description

In other words, it is an error if the pack procedure attempts to transfer array elements from the unpacked array that do not exist. For example given:

```
var
unpacked_value : array[1..10] of integer;
packed_value : packed array[21..30] of integer;
```

then

```
pack(unpacked_value, 2, packed_value);
```

means that the pack procedure should attempt to transfer ten elements from the unpacked array (since the packed array has ten elements), starting with the element two. So the pack procedure should transfer elements 2-11 from the unpacked array, but the unpacked array does not have an element 11.

Error Handling

This error is always reported.

18.30 Invalid use of unpack (1)

Official Description

6.6.5.4 For unpack, it is an error if the parameter of ordinal-type is not assignment compatible with the index-type of the unpacked array parameter.

Simplified Description

In other words, the parameter used to specify the first element of the unpacked array to receive elements transferred from the packed array must be assignment compatible with the index type of the unpacked array. Or to put it another way the first element of the unpacked array to receive elements transferred from the packed array must exist. For example given:

```
var
unpacked_value : array[1..10] of integer;
```

```
packed_value : packed array[21..30] of integer;
```

then

```
unpack(packed_value, unpacked_value, 0);
```

is an error since **0** is not assignment compatible with **1..10**. Or to put it another way element 0 of the unpacked array does not exist.

Error Handling

This error is always reported.

18.31 Invalid use of unpack (2)

Official Description

6.6.5.4 For unpack, it is an error if any of the components of the packed array are undefined.

Simplified Description

The unpack procedure transfers all of the elements of the packed array into the unpacked array, so it is an error if any of the elements of the packed array are undefined.

Error Handling

This error is never reported.

18.32 Invalid use of unpack (3)

Official Description

6.6.5.4 For unpack, it is an error if the index-type of the unpacked array is exceeded.

Simplified Description

In other words, it is an error if the unpack procedure attempts to transfer array elements from the packed array into elements of the unpacked array that do not exist. For example given:

```
var
unpacked_value : array[1..10] of integer;
packed_value : packed array[21..30] of integer;
```

then

```
unpack(packed_value, unpacked_value, 2);
```

means that the unpack procedure should attempt to transfer all ten elements from the packed array into the unpacked array starting at element 2 (i.e. transfer elements 21-30 from the packed array into elements 2-11 of the unpacked array), but the unpacked array does not have an element 11.

Error Handling

This error is always reported.

18.33 Square of large numbers

Official Description

6.6.6.2 Sqr(x) computes the square of **x**. It is an error if such a value does not exist.

Simplified Description

In other words, it is an error if sqr(x) is too large to be represented by the real type.

Error Handling

This error is always reported.

18.34 ln(x) and x is less than or equal to 0

Official Description

6.6.6.2 For ln(x), it is an error if **x** is not greater than zero.

Error Handling

This error is always reported.

18.35 Square root of negative number

Official Description

6.6.6.2 For sqrt(x), it is an error if **x** is negative.

Error Handling

This error is always reported.

18.36 Problems with trunc

Official Description

6.6.6.3 For trunc(x), the value of **trunc(x)** is such that if **x** is positive or zero then **0<=x-trunc(x)<1**; otherwise **-1<x-trunc(x)<=0**. It is an error if such a value does not exist.

Simplified Description

In other words, it is an error to use the trunc function to convert a value of real type into a value of integral type, if the real value is outside the range of integral values supported by Irie Pascal. For example

```
trunc(1.0e16)
```

is an error.

Error Handling

This error is never reported.

18.37 Problems with round

Official Description

6.6.6.3 For round(x), if **x** is positive or zero then **round(x)** is equivalent to **trunc(x+0.5)**, otherwise **round(x)** is equivalent to **trunc(x-0.5)**. It is an error if such a value does not exist.

Simplified Description

In other words, it is an error to use the round function to convert a value of real type into a value of integral type, if the real value is outside the range of integral values supported by Irie Pascal. For example

```
round(1.0e16)
```

is an error.

Error Handling

This error is never reported.

18.38 Problems with chr

Official Description

6.6.6.3 For chr(x), the function returns a result of char-type that is the value whose ordinal number is equal to the value of the expression **x** if such a character value exists. It is an error if such a character value does not exist.

Error Handling

This error is always reported.

18.39 Problems with succ

Official Description

6.6.6.4 For succ(x), the function yields a value whose ordinal number is one greater than that of **x**, if such a value exists. It is an error if such a value does not exist.

Error Handling

This error is reported if range checking is enabled.

18.40 Problems with pred

Official Description

6.6.6.4 For pred(x), the function yields a value whose ordinal number is one less than that of **x**, if such a value exists. It is an error if such a value does not exist.

Error Handling

This error is reported if range checking is enabled.

18.41 Using eof on undefined files

Official Description

6.6.6.5 When eof(f) is activated, it is an error if **f** is undefined.

Error Handling

This error is reported if I/O checking is enabled.

18.42 Using eoln on undefined files

Official Description

6.6.6.5 When eoln(f) is activated, it is an error if **f** is undefined.

Error Handling

This error is reported if I/O checking is enabled.

18.43 Using eoln at end-of-file

Official Description

6.6.6.5 When eoln(f) is activated, it is an error if eof(f) is true.

Error Handling

This error is never reported.

18.44 Using undefined variables

Official Description

6.7.1 An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error.

Simplified Description

In other words, it is an error to use a variable with an undefined value in an expression.

Error Handling

This error is reported only if checking for undefined values is enabled and the variable's type is one of the supported types (see Run-Time Project Options in the Irie Pascal User's Manual).

18.45 Real division by zero

Official Description

6.7.2.2 A term of the form **x/y** is an error if **y** is zero.

Simplified Description

In other words, for real division, it is an error if the denominator is zero.

Error Handling

This error is always reported.

18.46 Integer division by zero

Official Description

6.7.2.2 A term of the form **i div j** is an error if **j** is zero.

Simplified Description

In other words, for integer division, it is an error if the denominator is zero.

Error Handling

This error is always reported.

18.47 Modulus of zero or negative

Official Description

6.7.2.2 A term of the form **i mod j** is an error if **j** is zero or negative.

Simplified Description

In other words, for the modulus operation, it is an error if the right operand is zero or negative.

Error Handling

This error is always reported.

18.48 Integer overflow/underflow

Official Description

6.7.2.2 It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

Simplified Description

This appears to mean that it is an error if integer operations or functions overflow or underflow. So for example

```
maxint+maxint
```

should be an error since

```
maxint = 2,147,486,647
```

but due to integer overflow

```
maxint+maxint = -2
```

Error Handling

This error is never reported.

18.49 Returning undefined value from function

Official Description

6.7.3 It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation.

Simplified Description

In other words, it is an error if the result of a function call is undefined.

Error Handling

This error is reported if checking for undefined values is enabled and the type of the function's result is one of the supported types (see Run-Time Project Options in the Irie Pascal User's Manual).

18.50 Assigning out of range ordinal values

Official Description

6.8.2.2 For an assignment-statement, it is an error if the expression is of an ordinal-type whose value is not assignment-compatible with the type possessed by the variable or function-identifier.

Simplified Description

In other words, it is an error to assign an out-of-range ordinal value to a variable or a function result.

Error Handling

This error is reported if range checking is enabled.

18.51 Assigning out of range set values

Official Description

6.8.2.2 For an assignment-statement, it is an error if the expression is of a set-type whose value is not assignment-compatible with the type possessed by the variable.

Simplified Description

In other words, it is an error to assign an out-of-range set value to a variable. **NOTE:** The previous error said "... the variable or function-identifier" but this one just says "... a variable". The reason why "function-identifier" is not meantioned with this error description is that Standard Pascal does not allow set values to be assigned to function identifiers. Irie Pascal does allow set values to be assigned to function identifiers, and if range checking is enabled, does report assigning out-of-range set values to function identifiers.

Error Handling

This error is reported if range checking is enabled.

18.52 Non-matching case index

Official Description

6.8.3.5 For a case-statement, it is an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

Error Handling

This error is always reported.

NOTE: As an extension Irie Pascal supports the otherwise keyword which matches the case-index if none of the case-constants match the case-index. So technically in that situation this error is not reported.

18.53 For loops initial value out of range

Official Description

6.8.3.9 For a for-statement, it is an error if the value of the initial-value is not assignment-compatible with the type possessed by the control-variable, if the statement of the for-statement is executed.

Simplified Description

In other words, it is an error if the following two conditions are met:

- The initial value specified in a for statement is not in the range specified by the type of the control variable.
- At least one iteration of the for loop is executed.

For example given:

```
type num = 1..100;
var n : num;
```

then the following are errors:

```
for n := 0 to 100 do writeln(n);
for n := 101 downto 1 do writeln(n);
```

But the following are not errors since the for loops are never started (and therefore no assignments are made to **n**).

```
for n := 101 to 1 do writeln(n);
for n := 0 downto 100 do writeln(n);
```

Error Handling

This error is reported if range checking is enabled.

18.54 For loops final value out of range

Official Description

6.8.3.9 For a for-statement, it is an error if the value of the final-value is not assignment-compatible with the type possessed by the control-variable, if the statement of the for-statement is executed.

Simplified Description

In other words, it is an error if the following two conditions are met:

- The final value specified in a for statement is not in the range specified by the type of the control variable.
- At least one iteration of the for loop is executed.

For example given:

```
type num = 1..100;
var n : num;
```

then the following are errors:

```
for n := 1 to 101 do writeln(n);
for n := 100 downto 0 do writeln(n);
```

But the following are not errors since the for loops are never started (and therefore no assignments are made to the control variable **n**).

```
for n := 1 downto 101 do writeln(n);
for n := 100 to 0 do writeln(n);
```

Error Handling

This error is reported if range checking is enabled.

18.55 Reading invalid integer values

Official Description

6.9.1 On reading an integer from a text file, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-integer.

Simplified Description

In other words, it is an error if the program attempts to read an integer from a text file but the next characters to be read from the text file can not form an integer. For example if a text file contains the following

zzzzzzzzz

and you try to read in an integer value from this file it is an error.

Error Handling

This error is reported if I/O checking is enabled.

18.56 Reading out of range integer values

Official Description

6.9.1 On reading an integer from a text file, it is an error if the value of the signed-integer read is not assignment-compatible with the type possessed by the variable-access.

Simplified Description

In other words, it is an error to read an integer value from a text file into a variable, if the integer value is not assignment-compatible with the type possessed by the variable.

Error Handling

This error is reported if range checking is enabled.

18.57 Reading invalid numeric values

Official Description

6.9.1 On reading a number from a text file, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-number.

Simplified Description

In other words, it is an error if the program attempts to read an integer value or a real value from a text file but the next characters to be read from the text file can not form an integer value or real value. For example if a text file contains the following

```
zzzzzzzzz
```

and you try to read in an integer value or real value from this file it is an error.

Error Handling

This error is reported if I/O checking is enabled.

18.58 Reading when file is not open

Official Description

6.9.1 It is an error if the buffer-variable is undefined immediately prior to any use of read.

Simplified Description

This apparently means that it is an error to read from a file that is either not open or has been opened for writing.

Error Handling

This error is reported if I/O checking is enabled.

18.59 Writing with TotalWidth or FracDigits less than one

Official Description

6.9.3.1 On writing to a textfile, the values of TotalWidth and FracDigits are greater than equal to one; it is an error if either value is less than one.

Simplified Description

In other words, if you have

```
write(100:w)
```

or

```
write(100.23:w:f)
```

it is an error if **w** or **f** is less than one.

Error Handling

This error is reported if range checking is enabled.

18.60 Problems with program-parameters

Official Description

6.10 The execution of any action, operation, defined to operate on a variable, is an error if the variable is a program-parameter and, as a result of the binding of the program-parameter, the execution cannot be completed as defined.

Simplified Description

In other words, it is an error to use a program-parameter in a way that is not allowed because of its binding to an external entity.

Error Handling

The bindings of program parameters used by Irie Pascal do not prevent their use in any way and so this error can not occur.

18.61 Problems with conformant arrays

Official Description

6.6.3.8 For a conformant array, it is an error if the smallest or largest value specified by the index-type of the actual-parameter lies outside the closed interval specified by the index-type-specification of the conformant-array-schema.

Error Handling

Irie Pascal does not support conformant arrays so this error can not occur.

19.1 I/O error codes

Description

The following trappable error codes may be returned from various operations. If automatic error trapping is enabled (the default situation) then the compiler generates code to automatically check operations for trappable errors and report any errors detected. If you want more control on how errors are handled you can disable error trapping using the built-in procedure TrapErrors, and then use the function GetLastError to check the last trappable error that occured.

Value	Error Code	Cause
0	NoErrors	I/O operation completed successfully
1	FileEraseFailure	Attempt to erase a file failed
2	FileRenameFailure	Attempt to rename a file failed
3	FileIsUndefined	Attempt to operate on an undefined file
4	FileIsOpen	Using assign, erase, or rename on an open file
5	FileIsNotOpen	Attempt to perform I/O on a closed file
6	FileModeUndefined	THIS SHOULD NEVER OCCUR
7	FileNameTooLong	Attempt to give a file a name too long
8	FileNameNotDefined	Attempt to erase or rename a file with no name
9	FilePosFailure	Attempt to use pos on a file failed
10	FileSeekFailure	Attempt to use seek on a file failed
11	NotOutputFile	Attempt to write to a file not open for writing
12	UnexpectedEOF	Attempt to read past end-of-file
13	WriteFailure	Attempt to write to file failed
14	WidthTooLarge	Attempt to write a field too wide
15	NotInputFile	Attempt to read from a file not open for reading
16	ReadFailure	Attempt to read from file failed
17	ClosingFailure	Attempt to close a file failed
18	OpeningFailure	Attempt to open a file failed
19	FileBufferIsEmpty	Attempt to use put and the file-buffer is empty
20	ChDirFailure	Attempt to change directory failed
21	FileExpandFailure	Attempt to use fexpand failed
22	FileSplitFailure	Attempt to use fsplit failed
23	FileSizeFailure	Attempt to obtain size of file failed
24	FileOpenDirFailure	Attempt to open directory failed
25	FileReadDirFailure	Attempt to read directory entry failed

26	DirectoryIsUndefined	Attempt to access directory variable failed
27	FileCloseDirFailure	Attempt to close directory failed
28	FileRewindDirFailure	Attempt to rewind directory failed
29	FileMkDirFailure	Attempt to create directory failed
30	FileRmDirFailure	Attempt to remove directory failed
31	FileGetDateFailure	Attempt to get the date a file was last modified failed
32	FileGetModeFailure	Attempt to get the mode of a file failed
33	FileGetTimeFailure	Attempt to get the time a file was last modified failed
34	FileSetDateFailure	Attempt to set the date a file was last modified failed
35	FileSetTimeFailure	Attempt to set the time a file was last modified failed
36	FileRawReadFailure	Attempt to use rawread failed
37	FileRawWriteFailure	Attempt to use rawwrite failed
38	FileInternalFailure	THIS SHOULD NEVER OCCUR
39	DatabaseFailure	Attempt to access database failed
40	ObjectCreateFailure	Attempt to create object failed
41	ObjectGetOrFuncFailure	Attempt to get property to call method with no arguments failed
42	ObjectFuncFailure	Attempt to call function method failed
43	ObjectProcFailure	Attempt to call procedure method failed
43	ObjectPutFailure	Attempt to set object property failed
44	ObjectGenericFailure	Attempt to access generic object failed
45	MemoryAllocationFailure	Attempt to allocate memory failed
46	InvokeException	A generic object instance raised an exception

20.1 Grammar notation

In this manual the syntax of the Irie Pascal language is described using a context-free grammar. Context-free grammars are often used to describe the syntax of programming languages because they produce short and precise definitions. A context-free grammar breaks the syntax of a complex language into a number of simpler elements called non-terminal symbols. The syntax of each non-terminal symbol is defined by a production. Each production consists of a non-terminal symbol followed by some sort of assignment operator, and then followed by a sequence of special symbols, terminal symbols and non-terminal symbols. The special symbols are used to describe how the other symbols can be combined. The terminals symbols are literal text that can occur in the language being defined. The full context-free grammar takes the form of a sequence of productions for all the non-terminal symbols in the language.

The context-free grammar used in this manual has the following notation. The equal sign (i.e. =) is used as the assignment operator in productions to separate the non-terminal symbol being defined from its definition. The special symbols are |, {}, [], and (). The | symbol is used to separate alternatives (i.e. the grammar allows either the symbols on its left-hand side or the symbols on its right-hand side). The {} symbols are used to enclose symbols that can appear zero, one, or more times. The [] symbols are used to enclose optional symbols (i.e. symbols that can appear zero or one time). The () symbols are used to group symbols that are evaluated together. The **'terminal symbols'** appear in boldface and are enclosed in single quotation marks to distinguish them from non-terminal symbols. For example here are the productions that define the syntax for identifiers.

```
identifier =letter {letter |digit }

letter ='a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|
'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|
'u'|'v'|'w'|'x'|'y'|'z'|
'_'

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

These productions mean: An identifier is a letter followed by zero or more letters or digits. A letter is one of the listed characters (i.e. a-z) or an underscore. A digit is one of the decimal numerals.

The complete grammar for Irie Pascal is given in here.

20.2 Irie Pascal grammar

The start symbol for this grammar is **program**.

```
actual-parameter =expression |variable-access |
procedure-identifier |function-identifier

actual-parameter-list ='('actual-parameter {','actual-parameter }')'

adding-operator ='+'|'-'|'or'|'or_else'|'xor'

array-type ='array' [index-type-list ']' 'of'component-type

array-variable =variable-access

assignment-statement =assignment-statement-lhs ':='expression

assignment-statement-lhs =variable-access |function-identifier |property-designator

base-type =ordinal-type

binary-digit ='0'|'1'

binary-digit-sequence =binary-digit {binary-digit }

binary-integer ='%'binary-digit-sequence

block =declarative-part statement-part

boolean-expression =expression

buffer-variable =file-variable '^'|file-variable '@'

c-string-type ='cstring'[max-string-length ]

case-body =case-list-elements [[';']case-statement-completer ]|
case-statement-completer

case-constant =ordinal-constant

case-constant-list =case-specifier {','case-specifier }

case-index =ordinal-expression

case-list-element =case-constant-list ':'statement

case-list-elements =case-list-element {';'case-list-element }

case-specifier =case-constant ['..'case-constant ]

case-statement ='case'case-index case-body [';']'end'

case-statement-completer =('otherwise'|'else')statement-sequence

character-code =digit-sequence

character-literal ='''string-element-one '''|
'"'string-element-two '"'|
'#'character-code

component-type =type-denoter

component-variable =indexed-variable |field-designator

compound-statement ='begin'statement-sequence 'end'

conditional-statement =if-statement |case-statement
```

```
constant =[sign ]integer-number |
[sign ]real-number |
[sign ]constant-identifier |
character-literal |
string-literal

constant-definition =identifier '='constant

constant-definition-group ='const'constant-definition ';'{constant-definition ';'}

constant-identifier =identifier

control-variable =entire-variable

decimal-integer =digit-sequence

declaration-group =
label-declaration-group |
constant-definition-group |
type-definition-group |
variable-declaration-group |
function-declaration |
procedure-declaration

declarative-part ={declaration-group }

digit ='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

digit-sequence =digit {digit }

directive =forward-directive |external-directive

dllname =string-literal

domain-type =type-identifier

else-part ='else'statement

empty-statement =

empty-string =''''|'""'

entire-variable =variable-identifier

enumerated-constant =identifier

enumerated-constant-list =enumerated-constant {','enumerated-constant }

enumerated-type ='('enumerated-constant-list ')'

exponent ='e'

expression =shift-expression [relational-operator shift-expression ]

external-directive ='external'dllname ['name' '='name ]['stdcall'|'cdecl']

factor =[sign ]unsigned-constant |
[sign ]variable-access |
[sign ]'('expression ')'|
[sign ]function-designator |
[sign ]function-method-designator |
[sign ]'not'factor |
set-constructor

field-designator =record-variable '.'field-specifier |field-designator-identifier

field-designator-identifier =identifier
```

field-identifier =identifier

field-list =[
fixed-part ';'variant-part [';']|
fixed-part [';']|
variant-part [';']|
]

field-specifier =field-identifier

file-type ='**file**' '**of**'component-type

file-variable =variable-access

final-value =ordinal-expression

fixed-part =record-section {';'record-section }

for-statement ='**for**'control-variable '**:=**'initial-value ('**to**'|'**downto**')final-value
'**do**'statement

formal-parameter-list ='**(**'formal-parameter-section {';'formal-parameter-section }'**)**'

formal-parameter-section =value-parameter-specification |
variable-parameter-specification |
procedure-parameter-specification |
function-parameter-specification

forward-directive ='**forward**'

fractional-part =digit-sequence

function-block =block

function-declaration =
function-heading ';'directive |
function-identification ';'function-block |
function-heading ';'function-block

function-designator =function-identifier [actual-parameter-list]

function-heading ='**function**'identifier [formal-parameter-list]'**:**'result-type

function-identification ='**function**'function-identifier

function-identifier =identifier

function-method-designator =object-variable '**.**'function-method-identifier [actual-
parameter-list]

function-method-identifier =identifier

function-parameter-specification =function-heading

goto-statement ='**goto**'label

hex-digit =digit |'**a**'|'**b**'|'**c**'|'**d**'|'**e**'|'**f**'

hex-digit-sequence =hex-digit {hex-digit }

hexadecimal-integer ='**$**'hex-digit-sequence

identified-variable =pointer-variable '**^**'|pointer-variable '**@**'

identifier =letter {letter |digit }

identifier-list =identifier {'**,**'identifier }

if-statement ='**if**'boolean-expression '**then**'statement [else-part]

```
index-expression =expression

index-type =ordinal-type

index-type-list =index-type {','index-type }

indexed-variable =indexed-variable-array |indexed-variable-string

indexed-variable-array =array-variable '['index-expression {','index-expression }']'

indexed-variable-string =string-variable '['integral-expression ']'

initial-value =ordinal-expression

integer-number =decimal-integer |hexadecimal-integer |binary-integer

integral-constant =constant

integral-expression =expression

label =digit-sequence |identifier

label-declaration-group ='label'label {','label }';'

letter ='a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|
'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|
'u'|'v'|'w'|'x'|'y'|'z'|
'_'

list-type ='list' 'of'component-type

list-variable =variable-access

max-string-length ='['integral-constant ']'|'('integral-constant ')'

member-designator =ordinal-expression ['..'ordinal-expression ]

multiplying-operator ='*'|'/'|'div'|'mod'|'and'|'and_then'

name =string-literal

new-ordinal-type =enumerated-type |subrange-type

new-pointer-type ='^'domain-type |'@'domain-type

new-structured-type =
['packed']array-type |
['packed']record-type |
['packed']set-type |
['packed']file-type |
['packed']list-type |
object-type |
string-type

new-type =new-ordinal-type |new-structured-type |new-pointer-type

non-empty-string =
'''string-element-one string-element-one {string-element-one }'''|
'"'string-element-two string-element-two {string-element-two }'"'

object-type ='object'|'class'

object-variable =variable-access

ordinal-constant =constant

ordinal-expression =expression
```

```
ordinal-type =new-ordinal-type |ordinal-type-identifier

ordinal-type-identifier =identifier

pascal-string-type ='string'[max-string-length ]

pointer-variable =variable-access

printable-character =any character (including a space) that has a visual
representation.

procedure-block =block

procedure-declaration =
procedure-heading ';'directive |
procedure-identification ';'procedure-block |
procedure-heading ';'procedure-block

procedure-heading ='procedure'identifier [formal-parameter-list ]

procedure-identification ='procedure'procedure-identifier

procedure-identifier =identifier

procedure-method-identifier =identifier

procedure-method-specifier =object-variable '.'procedure-method-identifier

procedure-method-statement =procedure-method-specifier [actual-parameter-list ]

procedure-parameter-specification =procedure-heading

procedure-statement =procedure-identifier (
[actual-parameter-list ]|
read-parameter-list |readln-parameter-list |
write-parameter-list |writeln-parameter-list
)

program =program-heading ';'program-block

program-block =block

program-heading ='program'identifier ['('program-parameter-list ')']

program-parameter-list =identifier-list

property-designator =object-variable '.'property-specifier

property-identifier =identifier

property-specifier =property-identifier |'('property-string ')'

property-string =string-expression

read-parameter-list ='('[file-variable ',']variable-access {','variable-access }')'

readln-parameter-list =['('(file-variable |variable-access ){','variable-access }')']

real-number =
digit-sequence '.'fractional-part [exponent scale-factor ]|
digit-sequence exponent scale-factor

record-section =identifier-list ':'type-denoter

record-type ='record'field-list 'end'

record-variable =variable-access

record-variable-list =record-variable {';'record-variable }
```

```
relational-operator ='='|'<>'|'<'|'<='|'>'|'>='|'in'

repeat-statement ='repeat'statement-sequence 'until'boolean-expression

repetitive-statement =repeat-statement |while-statement |for-statement

result-type =type-identifier

scale-factor =[sign ]digit-sequence

selected-variable =list-variable '['index-expression {','index-expression }']'

set-constructor ='['[member-designator {','member-designator }]']'

set-type ='set' 'of'base-type

shift-expression =simple-expression [shift-operator simple-expression ]

shift-operator ='shl'|'shr'

sign ='-'|'+'

simple-expression =term {adding-operator term }

simple-statement =empty-statement |assignment-statement |
procedure-statement |procedure-method-statement |
goto-statement

statement =[label ':'](simple-statement |structured-statement )

statement-part =compound-statement

statement-sequence =statement {';'statement }

string-element-one =''''|printable-character

string-element-two ='""'|printable-character

string-expression =expression

string-literal =empty-string |non-empty-string

string-type =pascal-string-type |c-string-type

string-variable =variable-access

structured-statement =compound-statement |
conditional-statement |
repetitive-statement |
with-statement

subrange-type =constant '..'constant

tag-field =identifier

tag-type =ordinal-type-identifier

term =factor {multiplying-operator factor }

type-definition =identifier '='type-denoter

type-definition-group ='type'type-definition ';'{type-definition ';'}

type-denoter =type-identifier |new-type

type-identifier =identifier

unsigned-constant =integer-number |real-number |
character-literal |string-literal |constant-identifier |
'nil'

value-parameter-specification =identifier-list ':'type-identifier

variable-access =entire-variable |component-variable |identified-variable |
selected-variable |buffer-variable

variable-declaration =identifier-list ':'type-denoter

variable-declaration-group ='var'variable-declaration {';'variable-declaration }

variable-identifier =identifier

variable-parameter-specification ='var'identifier-list ':'type-identifier

variant =case-constant-list ':''('field-list ')'

variant-body =variant-list [[';']variant-part-completer ]|variant-part-completer

variant-list =variant {';'variant }

variant-part ='case'variant-selector 'of'variant-body

variant-part-completer =('otherwise'|'else')(field-list )

variant-selector =[tag-field ':']tag-type

while-statement ='while'boolean-expression 'do'statement

with-statement ='with'record-variable-list 'do'statement

write-parameter =expression [':'expression [':'expression ]]

write-parameter-list ='('[file-variable ',']write-parameter {','write-parameter }')'

writeln-parameter-list =['('(file-variable |write-parameter ){','write-parameter }
')']
```

The term *blank* is used in this manual to mean a space, tab, formfeed, or end-of-line character,and the term *blanks* means a sequence of *blank* characters.

A *directory variable* is a variable of the built-in type dir, that can be associated with a directory/folder on an external storage device, in such a way that the names of the files in the directory/folder can be read.

A *file* is a collection of values of a particular file type that is stored on an external storage device.

The *current file position* of a file is the position in the file that the next file operation will occur (i.e. the position of either the next value read from the file, or the next value to be changed in the file, or the next value added to the file).

The mode in which a file variable is opened specifies which operations can be performed on the file associated with the file variable, and also what happens to the file when the file variable is opened. If a file variable is opened in *read-only mode* then operations on the file associated with the file variable are allowed as follow:

- Values can be retrieved from the file.
- The existing values in the file can **not** be changed.
- New values can **not** be added to the file.

Also the file must already exist when the file variable is opened, it's contents are preserved, and the current file position is set to the beginning of the file.

The mode in which a file variable is opened specifies which operations can be performed on the file associated with the file variable, and also what happens to the file when the file variable is opened. If a file variable is opened in *write-only mode* then operations on the file associated with the file variable are allowed as follow:

- Values can **not** be retrieved from the file.
- The values in the file can be changed.
- New values can be added to the file.

Also if the file already exists when the file variable is opened then it's contents are deleted. If the file does not already exist when the file variable is opened then it is created, and is initially empty.

The mode in which a file variable is opened specifies which operations can be performed on the file associated with the file variable, and also what happens to the file when the file variable is opened. If a file variable is opened in *read/write mode* then operations on the file associated with the file variable are allowed as follow:

- Values can be retrieved from the file.
- The values in the file can be changed.
- New values can be added to the file.

Also if the file already exists when the file variable is opened then it's contents are deleted. If the file does not already exist when the file variable is opened then it is created, and is initially empty.

The mode in which a file variable is opened specifies which operations can be performed on the file associated with the file variable, and also what happens to the file when the file variable is opened. If a file variable is opened in *append mode* then operations on the file associated with the file variable are allowed as follow:

- Values can **not** be retrieved from the file.
- The values in the file can **not** be changed.
- New values can be added to the file.

Also if the file already exists when the file variable is opened then it's contents are preserved and the current file position is set to the end of the file. If the file does not already exist when the file variable is opened then it is created, and is initially empty.

A file's *modification date* is the date the file was last modified.

A file's *modification time* is the time the file was last modified.

What are write parameters?

Description

Write parameters are used, with the write and writeln procedures, to specify the values that these procedures should write to files.

Writing To Non-Text Files

If the file being written to is **not** a text file (i.e. **not** a file associated with a file variable of type text) then the *write parameters* are just expressions, and the values of these expressions are written without conversion to the file.

Writing To Text Files

If the file being written to is a text file (i.e. a file associated with a file variable of type text) then the *write parameters* can be made up of the following three items:

1. An expression specifying the value to be written to the file. **NOTE:** Non-string values are converted to string values before being written to the file, however the type of the *write parameter* is the type of the value before it was converted.
2. An optional expression of integral type that specifies the minimum width of the string value written to the file. If this item is not supplied then the minimum width used depends on the type of the *write parameter* as follows: If the *write parameter* has an integral type or boolean type then the default minimum width is eigth (8). If the *write parameter* is of real type then the default minimum width is nine (9). If the *write parameter* is of string type then the default minimum width is the length of the string. If the *write parameter* is of If the *write parameter* is of any other type then the default minimum width is one (1).
3. An optional expression of integral type that specifies the number of digits that follow the decimal point in the converted string (the value is converted into a fixed point representation). If this item is not supplied, and a value of real type is being converted, then the value is converted into an exponential representation. **NOTE:** This item only be used when the *write parameter* is of real type.

If the string value being written to the text file is shorter than the minimum width then it is padded with spaces on the left until it is the same length as the minimum width. If the string value being written to the file is longer than the minimum width then the entire string value is written to the file unless the *write parameter* is of string type or of boolean type in which case the string value is truncated on the right to the same length as the minimum width.

When a function or procedure is called, the calling code and the function or procedure being called must agree on three basic things:

1. The way parameters of different types are passed.
2. The way function return results are passed back to the calling code.
3. The way the responsibility for cleaning-up after the call is shared between the calling code and the function or procedure.

These agreements are called *calling conventions*.

Implementation-defined features may differ between implementations but **must** be consistent for any particular implementation.

Implementation-dependent features may differ between implementations and are **not** necessarily defined for any particular implementation.