# THE LANGUAGE PASCALINE

# REPORT VERSION 0.4

# Contents

# 1 Introduction

"Standards are great. Everyone should have one of their own" - Anon.

Pascaline is a formal statement of a language that was created over the period 1993 to 2008 in a series of extensions to ISO 7185 Pascal. The name "Pascaline" was chosen both to show that the language is designed to be %100 compatible with the original language, and to continue the language Pascal's tribute to Blaise Pascal. The Pascaline being Pascal's calculator, Pascaline the language is "The machine Pascal built", or the "Machine that runs Pascal".

In 2008, there are several defacto standards for an extended Pascal, and one official one, the ISO 10206 standard. I have been dissatisfied with these existing extensions for the simple reason that instead of extending Pascal, they are more akin to redesigns of the language. A good definition of what I mean by redesign would be the introduction of a feature that is designed to replace or duplicate a construct of original Pascal. In particular, the addition of the ability to coin pointer addresses to any variable and perform "type escapes" would remove at a stroke all of the type security Niklaus Wirth designed into Pascal.

At the same time, I wanted Pascaline to achieve a level of completeness that the language never achieved in its original form. My goal was not to create an instructional language, but a complete and practical implementation that could address current problems in computing without further extensions or special support packages.

To design Pascaline I have enumerated a set of goals for the language design:

- To be completely upward and downward compatible with ISO 7185 Pascal.
- To be a "logical extension" of original Pascal. That is, to extend Pascal using the same working theories and means as the original language, and poses no element that does not interoperate completely with the original language.
- To provide a reasonable upgrade to the language capability, that can be implemented using an existing standard compiler with minor effort compared to the original implementation of the compiler.
- To implement only features that could be implemented efficiently using existing computing hardware.

Pascaline was designed in a series of steps starting in 1993. For each feature, one or more proposals were made. Then, the proposals were evaluated, a winner chosen, and a test implementation in the compiler was made. Then, any adjustments required by the experience of actual use were performed. Every element in Pascaline is backed by a real implementation that is efficient and tested.

As for most modern languages, the major theme of Pascaline is for extending the language via libraries, objects and code reuse. Pascaline will, and should, "obsolete itself" by allowing user written extensions to such a point that the major thrust of development with the language would become that of developing libraries of functions to cover new areas in computer applications.

This very ability to extend the language also forms the basis of a new problem in standard implementation that, although it has existed from the time Pascal was originally designed in the 1970s,

has become ever more pressing. That is the definition of standard libraries and platforms. Towards this end, the Pascaline standard, as in the standard for most of today's languages, is divided into the base standard and a "platform", consisting of a series of standard libraries that handle common I/O and support problems in a machine and system independent way.

These libraries appear here as annexes. There is also a series of annexes covering issues such as character handling, string escapes and other "recommended practices" for Pascaline. The result should greatly aid the ability to write non-trivial Pascaline programs that are truly portable across machines, systems and implementations.

The second, but no less important theme in Pascaline, is parallel language execution. Pascaline is a thoroughly parallel language, and completes the idea that strong type security is a fundamental building block to parallel language implementation. Indeed, Pascaline now exists as one of the only languages that offers parallel execution security in a language, instead of being added in as a library with security left to the user.

As I have offered in the past for ISO 7185 Pascal, I extend the offer now, that I will evaluate Pascaline implementations for conformance to the Pascaline standard, no matter what the purpose of that implementation, public or private, profit or nonprofit. I only make the conditions that my access to the implementation be reasonable, that the authors provide me with a list of annexes that are complied with, and that an option exists in the implementation to enforce strict compliance with Pascaline regardless of any other extensions that might be present over and above Pascaline. This last is the same requirement that the ISO 7185 Pascal standard states.

This author further respectfully requests that the name "Pascaline" be applied only to an implementation that has been found to comply with the language specification here, and by the tests I provide free of charge or restriction. Further, unlike the ISO 7185 standard, I ask that no exceptions be allowed for the language (ISO 7185 5.1). A language may well comply with part of this specification, and be called a Pascal, or some other name. I only ask that it not be called "Pascaline" unless it can process the full language, aside from the annexes, without exception.

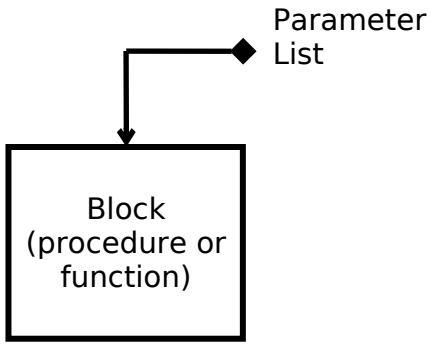
Scott A. Franco

July, 2008

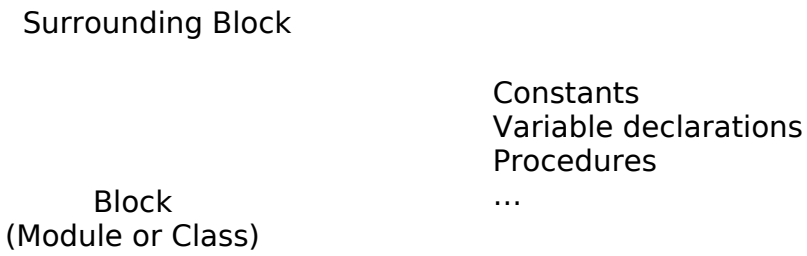
## 2 Summary of Pascaline extensions to Pascal

ISO 7185 Pascal defined a program as a series of nested blocks, one inside the other. The most basic block was the program block, which could contain any nested series of procedure and function blocks. Each block can contain a series of declarations containing labels, constants, types, variables, procedures and functions. Each block contains the code that executes the algorithms contained in the block.

The blocks of Pascal define a closed collection of these declarations with an interface that consists of a parameter list. In Pascaline terminology, this is a "top" interface, as the block communicates with the outside of the block via an interface at the "top" of the block:




In Pascaline, each modular block can also interface, with declarations in the surrounding block. This is referred to in Pascaline terminology as the "side" interface:




Each block can also import declarations from the surrounding block.

Pascal envisioned a program as a tower of blocks resting one atop the other. This paradigm is a good one, but views the program as a monolithic whole. The provision of fixed types without the ability to extend them with change also contributes to the model of a program created as a static structure of code.

Pascaline's main thrust is to add extensibility to Pascal, and this is done by greatly augmenting the methods to create side blocks. Pascaline views programs as a series of adjoining tiles:


In Pascaline, the program block is such an adjoining block, and adds several other block types that have this ability to export their declarations directly to other blocks. The primary of these is the **module**, which has all the powers of a program block, but adds the ability to specify both code that executes when the program starts, and code that executes when the program ends. In this way, modules appear as "service blocks" whose point is to provide constants, types, variables, procedures and functions to the program, along with a method both to set up such items as well as shut them down.

The **program** and **module** blocks form a group executing the same thread of execution for a program. Another module is the **process**, which defines a new thread of execution aside from the main program. Process modules cannot directly access program or module blocks, but the two can communicate via a monitor or channel block, which automatically implements the multitasking primitives needed to coordinate such an exchange. A share block gives a way to define constants, types, procedures and functions that are usable by any task without the overhead of multitask coordination.

Pascaline also defines a new level of block that is an intermediate between a so-called "global" block such as a program, module, monitor, channel, etc. Those blocks are static, with variables that are allocated for the duration of the program. These are natural outermost blocks, since any program ultimately is rooted in such program constructs. The class block fits between the level of global blocks and procedure and function blocks. A class block has the ability to share its declarations via the "side", but also be created dynamically or can be instantiated as part of the local variables of a block. Further, classes have the ability to be extended to any level. Any new class has the ability to be based on a previously defined class, and to have that class accepted as compatible with the base class.

Because classes define both declarations such as constants and types that have no allocation, as well as variables that do, the class must be instantiated either as part of the variable data, or via a reference. Any number of instances of a class may exist associated with such references. The instance of a class is called an object, and it is the set of the data associated with a class. The class contains the declaration of the format of the object it creates, and thus it is a "class" or "kind" for all of the objects created using it.

Because a class defines both a series of constants, types, variables, and also procedures and functions that can operate on those declarations, which are known as "methods", An object forms an instance of a module. Classes complete the idea of "object orientation" which dictates that data exists as paired with the procedures or functions that form its methods, needed to manipulate that data. Because classes

that inherit from each other also have references that are compatible, classes can extend each other to any number of levels to implement program concepts.

As a dynamic corollary to a module, classes also can have a separate thread of execution as a thread class, and perform as a tasking communications block as an **atom** or **stream** class.

To allow the static idea of parameter lists in Pascal to be extended, Pascaline implements the "overload" concept. Procedures, functions and methods can form "groups" under the same name that are differentiated by kind, type and number such that calls to such procedures and functions are sent to the instance that has the correct interface to operate on them. Built in expression operators can be overloaded, thus completing a full circle of data abstraction.

The concept of extendibility is further enhanced by the ability to override existing procedures, functions and methods. New modules and classes can override the previous meaning of them, and also extend them by performing new operations and calling the original definitions.

Pascaline has "container" types for arrays that do not specify an exact size. These types can be used to form a template to create such arrays of a runtime determined size at runtime. This allows procedures and functions to accept arbitrarily sized arrays to any dimension, and allows such arrays to be dynamically created as variables and pointer types as well.

The common case of an integer indexed array can be specified by a short form, which also underscores the idea of container types.

A line comment uses a single character, “!”, that allows the rest of the line to appear as a comment.

Pascaline introduces the break character, "_", for both identifiers and numbers. This aids readability for long identifiers and numbers.

Goto labels are freed from the restriction that they must appear as numbers, and can assume the same form as an identifier. Goto labels both retain their status as an interprocedure deep nested branching, but also allow for intermodular branching via a procedure or function call.

Pascaline implements the method of structured exception handling to handle deep nested returns. This allows code to be written that delivers exceptions to higher level code without needing knowledge of the surrounding code. This further enhances extendibility, and allows for complete replacement of “goto”s. It also introduces a standard method of error handling in Pascaline.

Constant expressions can be used wherever constants were used in Pascal. This makes it possible to use formulas for these constants instead of precalculated numbers.

Extended radixes allow direct specification of hexadecimal, octal or binary constants. These can also be read or written to files via I/O statements.

Boolean operations on integers are permitted, and a new operator, "xor", is implemented for both boolean and integer operands.

For procedures, functions and methods, two new parameter "modes" are implemented, the **view** and **out** modes. The **view** mode is identical to value parameter semantics, except that the parameter is protected from all modification. This makes certain compiler optimizations possible. The **out** mode is identical to **var** mode, but flags to the compiler that the parameter will be used only to pass out results.

Case statements now have an **else** clause, and ranges of case constants are possible.

Record case variant declarations can also use ranges of case constants.

**Write**/**writeln** can specify left justified fields, and a special mode allows the output of right padded strings in their natural length, and can specify any radix.

**Read**/**readln** can specify fields on read variables, and can specify literal strings to be matched to the input. A special mode allows the input of right padded strings in their natural length. They can also specify any radix.

A new declaration exists, **fixed**, which can be used anywhere a variable can, but cannot be modified. This allows the compile time specification of fixed tables, and ameliorates the need to create blocks of assignments at the start of a program.

Pascaline introduces a limited type conversion/restriction operation to convert between scalar types. This relieves the need to produce special handling to convert enumerated types to integer. It also introduces the ability to directly specify the precision needed within integer expressions, instead of always promoting such operations to the full size of integer. This allows more efficient numeric processing on small word size processors and arbitrary word length processors.

Pascaline standardizes a series of procedures and functions for files, such as binding to external file names, opening and closing a series of files, indexing within files, finding the length of a file, updating existing files, and appending to the end of such files, checking the existence of a file, and deleting and changing the name of a file.

New standard header parameters are introduced, including an error output, a list (or print) output, and a command line or file input.

The strict order of declarations from Pascal is relaxed in Pascaline. **label**, **const**, **type**, **var** and fixed declarations can occur in any order. This aids in the modular structure of Pascaline.

When a forwarded procedure, function or method appears as the actual declaration, the parameter list can be repeated. It is checked for congruence with the original. This allows such declarations to be created by cut and paste, and is more readable than the original method of having the actual declarations far from their forwarded declarations.

Procedures and functions can be declared as static, or non-recursive. This allows the creation of more efficient code on some processors.

Asserts are implemented, allowing the incorporation of runtime checks for code being debugged that can be removed without modifying the source. These can be annotated with output messages.

Pascaline allows "subrange" types to be created that are larger than the natural Pascal range of an integer (-**maxint**..**maxint**). This allows an implementation to implement types that utilize double or more precision while taking longer to perform them, so called "extended range" types. It also implements a set of predefined types that give an implementation defined set of unsigned and signed extended range types.

Besides standard reals, there is a type that is smaller than the standard real, and a type that is larger than the standard real. There are new constants that give the maximum values in each real type.

There is a predefined constant for the maximum character value in an implementation.

Vector or matrix math is supported.

Saturated math is supported, both for integers and for vector and matrix operations.

A way to specify a function result that obeys the rule of single entry/single exit with the result formed at the end. This is also required for operator overloads.

Function results are extended to allow most types, including structured, to be returned as a result.

Properties specify a program object that appears as a variable, but has its read and write actions completely program defined. Properties are also a building block to advanced multitasking structures.

A list of new common types, such as strings, vectors and matricies are standard definitions.

The object model is extended to all types, even integers, files and other standard types, and methods using those objects are defined as alternatives to the procedural versions. Any type, including standard types, can be extended by user defined classes.

Besides the extensions to the base Pascal language, Pascaline defines a set of optional extension modules that define the Pascaline "platform". This is a nod to the fact that the set of support calls for an implementation make as much difference to portability for a program as the base language does.

## 3 EBNF and syntax used in this document

For the purpose of describing syntax elements of Pascaline, the EBNF or Extended Backus-Naur Format as used in the ISO 7185 standard is used. The syntax that appears here consists of the syntax elements from the ISO 7185 standard as modified for Pascaline use.

name = syntax-description .

Describes the syntax expansion of the syntax element by name of "name", which is terminated by '.'.

a | b

Either construct a or constructor b may appear, but not both.

{ a }

Construct a is repeated 0 or more times.

[ a ]

Construct a is repeated 0 or 1 times (it is optional).

'abc'

The characters "abc" appear literally.

( a b )

The elements a and b are grouped together.

The syntax expansions that appear in this document mirror the same by name in the ISO 7185 document. If a syntax element by name matches a name used in the document, and is different from the one contained there, then it represents a Pascaline extension to ISO 7185 Pascal, and replaces the original syntax definition.

Note the ISO 7185 use of ">" or "alternate" is not used here, because Pascaline does not use the "level 1" extensions of ISO 7185. The syntax for the level 1 extensions was removed.

Annex A contains a full syntax for Pascaline.

# 4 Relationship to ISO 7185

## 4.1 Pascaline as a series of extensions

The Pascaline standard accepts the entire language defined in the standard defined in ISO 7185 such that the set of features defined by Pascaline qualify as extensions under ISO 7185 3.2 “Extension”.

## 4.2 Additional reserved word-symbols

As allowed in ISO 7185 3.2 “Extension”, the following additional spellings of identifiers are prohibited because of their use as word-symbols in the Pascaline standard:

| | | | | |
|---|---|---|---|---|
| **forward** | **module** | **uses** | **private** | **external** |
| **view** | **fixed** | **process** | **monitor** | **share** |
| **class** | **is** | **xor** | **overload** | **override** |
| **reference** | **joins** | **static** | **inherited** | **self** |
| **virtual** | **try** | **except** | **extends** | **on** |
| **result** | **operator** | **start** | **task** | **atom** |
| **property** | **channel** | **stream** | **out** | |

These word-symbols are new word-symbols defined by Pascaline. The fact that certain identifiers are prohibited in Pascal may cause ISO 7185 compliant programs to fail to compile for this reason. To use a Pascaline implementation for such programs, one of two methods are used:

1. The ISO 7185 compliance switch is enabled (see ISO 7185 5.1 “Processors” note 2).
2. The identifiers in the program that overlap the above list are changed.

## 4.3 Character escapes

Annex E “Character Escapes”, if implemented by the target system, can affect the behavior of character strings in an ISO 7185 implementation. Although ISO 7185 does not specify the character set or contents of strings in a complying program, it is reasonable for a given program to assume that all visible characters are treated equally.

If the program contains the character ‘\’, the escape sequence introduction character, this will be treated differently than other characters. To use a Pascaline implementation that implements Annex E for such programs, one of three methods are used:

1. The ISO 7185 compliance switch is enabled (see ISO 7185 5.1 “Processors” note 2).
2. A switch that enables or disables character escapes is set to disable.
3. All instances of ‘\’ within character strings are changed to ‘\\’.

## 4.4 Compliance with ISO 7185

ISO 7185 5.2 “Processors” allows an implementation to be considered compliant if it is accompanied by a list of the requirements for which it does not comply. Pascaline specifically does not allow such exceptions. A processor is Pascaline compliant only if it has the following characteristics:

1. The complete requirements of ISO 7185 are followed, without exception.
2. The complete requirements of Pascaline, in this document, are followed without exception.

That is, the “list of requirements not complied with” from ISO 7185 5.2 “processors” is neither allowed as a starting point for Pascaline, nor carried forward into the language Pascaline.

### 4.5 ISO 7185 level 0 only

Pascaline is compliant with “level 0” Pascal only, as allowed for by the ISO 7185 standard (ISO 7185 5 “Compliance”). Conformant arrays are not specified in Pascaline, nor are they specified as an option under Pascaline. However, such conformant arrays could well be implemented as an extension to Pascaline.

### 4.6 Extensions to Pascaline

Extensions to Pascaline are changes to language features in this standard that do not cause a program complying with this standard to fail to compile, interpret, run or otherwise function with the exception that one or more additional spellings of symbols may be reserved to the processor. That is, additional reserved word-symbols may be defined.

An example of an extension that does not comply with this standard is a modal change such as redefining the functionality of the “mod” operator. This would cause otherwise complying programs to fail to function.

### 4.7 Compliance statement

Complying processors with Pascaline shall issue the statement:

<This processor> complies with the requirements of Pascaline version X.X [and the annexes A, B, C…]

Where:

| | |
|---|---|
| <This processor> | Is the name of the complying processor |
| X.X | Is the version number of the Pascaline specification complied with. |
| [and the annexes A, B, C..] | Is an optional list of the annexes also compiled with] |

There is no minimum requirement for the number of annexes complied with. The processor may comply with any number from zero to all of them. Just as compliance with the basic Pascaline standard implies that a program will or will not run, the presence or absence of an annex a program relies on may cause it to compile and run, or fail to do so.

### 4.8 Compliance switch

If the processor complying with this standard contains extensions (defined in 4.6 “Extensions to Pascaline”), it shall contain a switch for compliance with Pascaline with the following characteristics:

On:

1. Causes the Pascaline compliance statement to be issued (4.7 “Compliance statement”).

2. Causes any restriction on spelling of identifiers to be removed (no additional reserved words beyond those of Pascaline).
3. Causes any extension to the Pascaline standard to be removed.

Off:

1. Causes the Pascaline compliance statement to be removed (4.7 “Compliance statement”).
2. Enables any extensions to the Pascaline standard.
3. Does not cause any program complying with the Pascaline standard to fail to compile, interpret, run or otherwise function, with the exception of restriction of spelling of identifiers (additional reserved words).

Note that if the processor contains extensions that do not comply with the Pascaline definition of extensions (4.6 “Extensions to Pascaline”), the processor will, by definition, not comply with the “Off” switch requirement above.

## 4.9 Similarities with the ISO 7185 standard document

In general much of the form and manner of the original ISO 7185 standard document was followed in this document. However, in one case there is a distinct difference. The original ISO 7185 document does not distinguish between errors at compile time and at runtime, which sometimes overlap. In this standard, all runtime errors are represented as exceptions, and a list of exceptions appears as Annex B.

The use of “equivalent code” occurs many times in this document. That is, a built-in or standard operation or statement is explained in terms of other statements in ISO 7185 Pascal or Pascaline. This creates a concrete and exact explanation of standard actions.

## 5 Relationship to the Pascal-P6 series compiler

In 1972 A project was begun to create a portable compiler that accepted and processed a subset of the original Jensen and Wirth Pascal standard. This compiler went through several versions, and ended as the Pascal-P4 compiler. In its final iteration, it still a subset compiler, with several features of J&W left out. This compiler was extensively documented by both the Zurich originators and in “Pascal Implementation: the P4 Compiler” [S. Pemberton and M.C. Daniels].

In 1982, the ISO 7185 standard was issued, and this was accompanied by the ISO 7185 compiler in A model Implementation of Standard Pascal [Welsh], and by a BSI (British Standards Institute) test suite designed to fully test existing compilers.

In 2009 a project was begun to improve the original Zurich Pascal-P compiler to accept and process the ISO 7185 Pascal standard. This resulted in Pascal-P5, which replaces the model implementation. It was accompanied by an extensive document and by a new test suite designed to replace the BSI test suite, which is no longer available.

In 2017 a project was begun to implement the Pascaline language as an increment to the Pascal-P5 compiler, and create a new test suite to verify the result, and also a new working document for the compiler.

The Pascal-P6 implementation provides a freely available, public domain example implementation of Pascaline whose test suite is also freely available. This can be used as an example for a new Pascaline implementation, as a starting point for such an implementation, and/or as a source of tests for such an implementation, or simply as a reasonable implementation in its own right.

Pascal-P6 and its associated tests provide the proving system for the Pascaline language. It is expected that when the Pascal-P6 implementation is complete and tested, that this standard will reach a 1.0 version.

# 6 Language extensions for Pascaline

## 6.1 Word-symbols

word-symbol = 'and' | 'array' | 'begin' | 'case' | 'const' | 'div' | 'do' | 'downto' | 'else' | 'end' | 'file' | 'for' | 'function' | 'goto' | 'if' | 'in' | 'label' | 'mod' | 'nil' | 'not' | 'of' | 'or' | ‘xor’ | 'packed' | 'procedure' | 'program' | 'record' | 'repeat' | 'set' | 'then' | 'to' | 'type' | 'until' | 'var' | 'while' | 'with' | 'forward' | 'module' | 'uses' | 'private' | 'external' | 'view' | 'fixed' | 'process' | 'task' | 'monitor' | ' share' | 'class' | 'is' | 'overload' | 'override' | 'reference | 'joins' | 'static' | 'inherited' | 'self' | virtual' | 'try' | 'except' | 'extends' | ‘on’ | ‘result’ | ‘operator’ | ‘out’ | ‘property’ | ‘channel’ | ‘stream’.

Notes:

1. The directives "external" and "forward" in ISO 7185 Pascal are promoted to word-symbols in Pascaline.
2. The function of the directive “external” in Pascaline is not defined, just as it was not defined in ISO 7185 Pascal.

## 6.2 Special symbols

special-symbol = '+' | '-' | '*' | '/' | '=' | '<' | '>' | '[' | ']' | '.' | ',' | ':' | ';' | '^' | '(' | ')' | '<>' | '<=' | '>=' | ':=' | '..' | '(.' | '.)' | '@' | word-symbol .

Note that the alternative symbols ‘(.’, ‘.)’, and ‘@’ were optional in ISO 7185 Pascal, and remain optional in Pascaline.

## 6.3 Comments

The character ‘!’ introduces a “Line Comment”. All characters up to and including the next end of line are ignored. This allows a comment form that is short, and automatically terminated by the end of line:

```
!
! A simple program
!
program p;

begin

   writeln(‘Hello, world’) ! print to console

end.
```

Notes:

1. The ‘!’ character must appear between Pascaline symbols and not within any symbol.
2. Any other comment symbols within a line comment are ignored, ‘{‘, ‘}’, ‘(*’, or ‘*)’.

## 6.4 Identifiers

Identifiers in Pascaline are identical to ISO 7185 Pascal, with the addition of the break character '_'. An identifier can start with any of 'a'..'z' or '_', and continue with 'a'..'z', '_' and '0'..'9'. As in ISO 7185 Pascal, identifiers are not case sensitive.

identifier = letter | '_' { letter | digit | '_' }.

Example identifiers:

one_more_time

_last_time

## 6.5 Labels

Labels, used for **goto** purposes, can use the same format as identifiers (see 6.4 “Identifiers”)under Pascaline. The original "apparent value" numeric labels of ISO 7185 Pascal are accepted as well.

label = digit-sequence | identifier .

Example label:

```
program p;

label exit;

{ declarations }

begin

   goto exit

   { statements to be skipped }

   exit:

end.
```

## 6.6 Numeric constants

unsigned-integer = decimal-integer | hex-integer | octal-integer | binary-integer .

decimal-integer = digit-sequence .

hex-integer = '$' hex-digit-sequence .

octal-integer = '&' octal-digit-sequence .

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

digit-sequence = digit { digit } .

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

octal-digit-sequence = octal-digit { octal-digit }

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

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

hex-digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | '_' .

octal-digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '_' .

binary-digit = '0' | '1' | '_' .

Both standard integer and real specifications are available. In addition, three "radix specifier" formats are available:

$1234 - Specifies hexadecimal format (base 16)

&1234 - Specifies octal format (base 8)

%1010 - Specifies binary format (base 2)

Each alternative format is specified with a "radix introduction" character. These formats can be specified anywhere the Pascaline construct unsigned-constant is specified.

Pascaline accepts a "break character" within numeric constants. The "_" character can be used anywhere within a number:

123_456_789

It cannot be used as the first character of a number, which would make it an identifier.

The meaning of the number is considered without the break characters. The number:

1_2_3

is equivalent to:

123

Break characters can be used in any radix. They are useful for grouping digits so that the result is more readable:

12_334_222 { marked in thousands }

$5566_1212 { marked in 16 bit sections }

## 6.7 Constant expressions

constant = [ sign ] constant-term { adding-operator constant-term } .

constant-term = constant-factor { multiplying-operator constant-factor } .

constant-factor = '(' constant ')' | 'not' constant-factor | constant-set-constuctor | character-string | constant-identifier | unsigned-number .

constant-set-constructor = '[' [ constant-member-designator { ', ' constant-member-designator } ] ']' .

constant-member-designator = constant [ '... ' constant ] .

multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' .

adding-operator = '+' | '-' | 'or' | 'xor' .

Whenever a constant appears in Pascal, Pascaline is able to accept a constant expression. Constant expressions have a syntax that is similar, but not identical to standard expressions in Pascal. Constant expressions can only operate on other constants or constant expressions, and cannot include variables.

Note: The operator **xor** is defined in 6.8 “Boolean integer operations”.

## 6.8 Boolean integer operations

expression = simple-expression [ relational-operator simple-expression ] .

simple-expression = [ sign ] term { adding-operator term } .

term = factor { multiplying-operator factor } .

factor = variable-access | unsigned-constant | function-designator | type-identifier '(' expression ')' | set-constructor | '(' expression ')' | 'not' factor .

multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' .

adding-operator = '+' | '-' | 'or' | 'xor' .

Besides the use of **and**, **or** and **not** on Booleans in Pascal, Pascaline allows the use of **and**, **or** and **not** with integer operands. The result is the bitwise 'and', ‘or’ or ‘not’of the bits in the integer.

Pascaline defines a new operator, **xor**, which has the same precedence as **and**. It gives the bitwise exclusive or of integers. It can also be used on boolean types, but in that case is equivalent to the operation a <> b.

Boolean operations on negative values are not defined, and may be treated as errors, caught at either compile time or run time, by the processor.

The boolean integer **or** operation is equivalent to:

```
function bor(a, b: integer): integer;

var i, r, p: integer;

begin

   if (a < 0) or (b < 0) throw(NegativeInteger);
   r := 0; { clear result }
   p := 1; { set 1st power }
   i := maxint; { set maximium positive number }
   while i <> 0 do begin

      if odd(a) or odd(b) then r := r+p; { add in power }
      a := a div 2; { set next bits of operands }
      b := b div 2;
      i := i div 2; { count bits }
      if i > 0 then p := p*2 { find next power }

   end;
   bor := r { return result }

end;
```

The boolean integer **and** operation is equivalent to:

```
function band(a, b: integer): integer;

var i, r, p: integer;

begin

   if (a < 0) or (b < 0) throw(NegativeInteger);
   r := 0; { clear result }
   p := 1; { set 1st power }
   i := maxint; { set maximum positive number }
   while i <> 0 do begin

      if odd(a) and odd(b) then r := r+p; { add in power }
      a := a div 2; { set next bits of operands }
      b := b div 2;
      i := i div 2; { count bits }
      if i > 0 then p := p*2 { find next power }

   end;
   band := r { return result }

end;
```

The boolean integer **xor** operation is equivalent to:

```
function bxor(a, b: integer): integer;

var i, r, p: integer;

begin

   if (a < 0) or (b < 0) throw(NegativeInteger);
   r := 0; { clear result }
   p := 1; { set 1st power }
   i := maxint; { set maximum positive number }
   while i <> 0 do begin

      if odd(a) <> odd(b) then r := r+p; { add in power }
      a := a div 2; { set next bits of operands }
      b := b div 2;
      i := i div 2; { count bits }
      if i > 0 then p := p*2 { find next power }

   end;
   bxor := r { return result }

end;
```

## 6.9 view and out parameters

formal-parameter-list = '(' formal-parameter-section { ';' formal-parameter-section } ')' .

formal-parameter-section = value-parameter-specification | variable-parameter-specification | view-parameter-specification | out-parameter-specification | procedural-parameter-specification | functional-parameter-specification .

value-parameter-specification = identifier-list ':' type-identifier .

variable-parameter-specification = 'var' identifier-list ':' type-identifier .

view-parameter-specification = 'view' identifier-list ':' type-identifier .

out-parameter-specification = 'out' identifier-list ':' type-identifier .

procedural-parameter-specification = procedure-heading .

functional-parameter-specification = function-heading .

A parameter to a procedure or function has a type and a "mode", that indicates the method of its passage.

Besides the Pascal parameter modes of **var** and value, Pascaline adds a mode introduced by **view**:

```
program p(output);

type a = packed array [1..40] of char;

procedure x(view b: a);

begin

   ! use, but don’t change, b
   write('The string is: ', a)

end;

begin
end.
```

**view** parameters have the same characteristics as value parameters, and can be treated identically to value parameters. A **view** parameter cannot be modified or "threatened" in the procedure or function it belongs to. The meaning of "threatened" is the same as for **for** index variables of ISO 7185 6.8.3.9, and means that the parameter cannot be assigned, used as an index in a **for** loop, or passed to another routine as a **var** parameter.

**view** parameters enable the underlying implementation to create more efficient code in many cases. If the above example was:

```
program p;

type a = packed array [1..40] of char;

procedure x(b: a);

begin

   ! use, but don’t change, b
   write('The string is: ', a)

end;

begin
end.
```

The code would have to copy the actual parameter to **b**, which can be very inefficient for large arrays. The use of the **view** parameter allows the system to treat the parameter as a **var** parameter, but allow any expression to be passed as a value parameter would.

An **out** parameter is functionally identical to a **var** parameter except that it indicates that the parameter will be used only to return results to the caller, and that the contents of the variable passed will not be initialized on entry. This allows the processor to check for accesses to the variable before it has been assigned.

```
program p;

type a = packed array [1..100] of char;

procedure x(out b: a);

begin

   { set new contents for the array }
   for i := 1 to 100 do b[i] := i+10

end;

begin
end.
```

### 6.10 Extended case statements

case-statement = 'case' case-index 'of' case-list-element { ';' case-list-element } [ ';' ] 'end' .

case-list-element = case-list-statement | case-list-default .

case-list-statement = case-constant-list ':' statement .

case-list-default = 'else' statement .

case-constant-list = case-constant-range { ',' case-constant-range } .

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

case-index = expression .

A case statement can feature an **else** clause at the end of all case selects for a given case statement:

```
program p(input, output);

var y: integer;

begin

   readln(y);
   case y of

      1: write('one');
      2: write('two')
      else write('value unknown')

   end

end.
```

Is equivalent to:

```
program p(input, output);

var y: integer;

begin

   readln(y);
   if (y <= 0) or (y > 2) then write('value unknown')
   else case y of

      1: write('one');
      2: write('two')

   end

end.
```

The statement indicated as **else** will be executed if the case select value does not match any of the values in the case list.

A case select value can appear as a range of values:

```
program x;

var y: integer;

begin

   case x of

      1: write('one');
      2: write('two');
      3..5: write('three to five')

   end

end.
```

Is equivalent to:

```
program p;

var y: integer;

begin

   { statements that set y }

   case y of

      1: write('one');
      2: write('two');
      3, 4, 5: write('three to five')

   end

end.
```

## 6.11 Variant record case ranges

The case constants on a variant record specification can appear as ranges:

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

field-list = [ ( fixed-part [ ';' variant-part ] | variant-part ) [ ';' ] ] .

variant-part = 'case' variant-selector 'of' variant { ';' variant } .

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

case-constant-list = case-constant-range { ',' case-constant-range } .

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

```
program p;

type sr = 1..5;

var r: record

          i: integer;
          case s: sr of

             1: (q: integer; b: boolean);
             2: (z: real; t: array 10 of integer);
             3..5: (o: set of char; j: integer)

       end;

begin
end.
```

Is equivalent to:

```
program p;

type sr = 1..5;

var r: record

          i: integer;
          case s: sr of

             1: (q: integer; b: boolean);
             2: (z: real; t: array 10 of integer);
             3, 4, 5: (o: set of char; j: integer)

       end;

begin
end.
```

Notes:

1. For each case range, the starting constant must be less than or equal to the ending constant.
2. The ISO 7185 Pascal requirement that all values of the tagfield be present remains.

## 6.12 Array type shorthand

array-type = 'array' [ dimension-specifier ] 'of' component-type .

dimension-specifier = index-specifier | range-specifier .

index-specifier = '[' index-type { ',' index-type } ']' .

range-specifier = unsigned-integer { ',' unsigned-integer } .

Pascaline features a special declaration format for the most common case of integer indexed arrays:

```
type x = packed array 10 of char;
type y = array 20,10 of integer;
```

are equivalent to:

```
type x = packed array [1..10] of char;
type y = array [1..20,1..10] of integer;
```

Denoting an array index by size, instead of by an index specification, uses a different syntax, without the '[' and ']' characters, to make it clear what kind of array specification is being used.

## 6.13 Container arrays

array-type = 'array' [ dimension-specifier ] 'of' component-type .

The specification of only fixed arrays in ISO 7185 Pascal caused several practical issues with the language. Pascaline has the concept of "container arrays", which are array types specified without index specifications.

```
type a = array of integer;
```

Container arrays are a "template type" which cannot be used to directly specify a static variable without qualification. For example:

```
{ incorrect example }

program p;

type a = array of integer;

var b: a;

begin
end.
```

Would not be correct because type **a** does not contain enough information, namely the size of the array, to define the static variable **b**.

The power of container arrays is their ability to be bound to a fully defined static array at a time later than when the program is compiled. This can happen in three different ways:

1. When a parameter is accepted for a procedure or function.
2. As a result of the creation of an anonymous array in dynamic storage.
3. By use of a parameterized variable declaration (see 6.14 “Parameterized variables”).

It is important to understand that Pascaline does not specify what are called "dynamic arrays", which is the ability to resize an array at runtime. Pascaline simply provides the means to create fixed arrays of any size at runtime and bind to them, or to create generate purpose procedures or functions that can bind to any size array at runtime. This distinction may seem to be unnecessary, but in practice it dramatically simplifies implementation of the concept.

Container types can have multiple index levels:

```
type a = array of array of array of boolean;
```

Container array types are compatible with other container and fixed array types that:

1. Have the same base types.
2. Have the same packed/unpacked status.
3. Have the same number of index levels.

Thus, the following types are compatible:

```
type a = packed array [1..100] of char;
```

and

```
type b = packed array of char;
```

```
type a = array [1..10, 1..100] of integer;
```

and

```
type b = array of array of integer;
```

A container array can be used for a procedure or function parameter:

```
type string = packed array of char;

procedure x(var s: string);
```

And can use any mode.

```
program p;

type a = packed array [1..40] of char;

procedure b(c: a; var d: a; view e: a);

begin

   writeln('Copy is: ', c, ' actual variable is: ', d,
           ' view is: ', e)

end;

begin
end.
```

As with ISO 7185 Pascal array parameters, if the parameter is passed by value, there can be considerable expense associated with copying the array to private location for the procedure or function.

When container arrays are used as **var** parameters, they are simply serving as holders for an existing array. In this case, the array was created elsewhere in the code and the parameter simply points to it. In the case of value parameters, the array is created anew in a private location, and the existing array copied to it. In both cases, the information needed to complete the template for the array exists elsewhere.

Another way to allocate a container array is to use **new** to allocate it dynamically:

```
program p;

type ap = ^a;
     a  = packed array of char;

var b: ap;

begin

   new(b, 10);
   b := 'hi there  ';

end.
```

The call of **new** must specify the size of all indices, in the same order as the indices appear in the declaration. The parameters can be any expression. The call of **dispose** does not specify the indices of the array.

Container arrays have the property that they are free of index typing. Container arrays are considered to be compatible with other arrays (standard or container) that have the same number of base elements and packing status.

This means that the following assignments are possible:

```
program p;

type a = packed array [20..120] of char;
     b = array of char;
     c = ^b;

var q: a;
    x: c;

begin

   new(x, 100);

   x^ := q;

   q := x^;

end.
```

It does not matter what the first or last element of the array is. When elements are copied from or to a compatible container, or when a container is used to reference a standard array, the elements are matched up from the first to the last without regard to the exact number of the index used. In the last example, q[20] and x[1] refer to the same element. The equivalent of the above copy operation is:

```
program p;

type a = packed array [20..120] of char;
     b = array of char;
     c = ^b;

var q: a;
    x: c;

begin

   new(x, 100);
   for i := 1 to 100 do x^[i] := q[i+20-1];
   for i := 1 to 100 do q[i+20-1] := x^[i]

end.
```

When a container array is indexed, the method of specifying its indices is always the integers 1 to n, where n is last element of the array. The last element of a container array, which happens to also be its length, can be found by the predefined function:

```
max(a, l);
```

Where **a** is the container array, and **l** is the indexing level. For example:

```
program p;

type a = array of array of array of integer;
     b = ^a;

var c: b;
    i: integer;

begin

   new(c, 10, 20, 30);

   i := max(c, 2)

end.
```

Finds the second level index length of **b**, which is 20.

Note that the array argument to max must be a variable reference.

Specifying a level for **max** is optional. The function call:

```
max(a);
```

is equivalent to:

```
max(a, 1);
```

Container arrays make it possible to have a type that represents a string, or packed array of characters with a starting index of 1:

```
type string = packed array of char;
```

This is a standard system definition in Pascaline (see 6.38 ”Common Types”). In addition, a pointer to a string is also standard:

```
type pstring = ^string;

program p(output);

var s(20): string;
    sp: pstring;

begin

   new(sp, 10);
   s := ‘hello               ‘;
   sp^ := ‘john      ‘

end.
```

## 6.14 Parameterized Variables

Although containers cannot be directly used as the type of a variable, or part of a variable, they can be used for the type of a parameterized variable declaration:

initialized-identifier = identifier [ ‘(‘ expression { ‘,’ expression ‘)’ } ] .

variable-identifier-list = initialized-identifier { ‘,’ initialized-identifier } .

variable-declaration = variable-identifier-list ':' type-denoter .

variable-declaration-part = [ 'var' variable-declaration ';' { variable-declaration ';' } ] .

```
program p;

type z = array of integer;

procedure q(v: z);

var a(max(v)): array of integer;
    i:         integer;

begin

   for i := 1 to max(v) do a[i] := v[i]+1

end;

begin
end.
```

Variables which are declared using containers use a parameterized form that takes one or more expressions on the left side, which are used to fulfill the indices that appear in the type of the variable on the right side.

The parameter expressions must be integer.

Each expression will be used to match an index in the type. If the type is formed from a nested declaration, the indices are processed from the outermost declaration to innermost. If multiple dimension arrays exist, the indices appear from major to minor.

Every index in the type must be matched. If there are more or fewer index expressions than there are indices in the type, it is an error.

The expression used for an index parameter is not limited. Variables, parameters and functions can be used, but all of the elements used must be fully defined. The variables defined in the current block cannot be used since they are not initialized by definition.

The order of initializer execution is not specified. If functions have side effects, this can cause them to fail.

The evaluation of the index parameters is performed before the code in the block for which the variable is declared is executed. The indices do not vary during the execution of the declaring block. This means the following variant of the above code would function correctly:

```
program p;

type z = array of integer;

procedure q(l: integer; v: z);

var a(l): array of integer;
    i:    integer;

begin

   l := 5;
   for i := 1 to l do a[i] := v[i]+1

end;

var r(10): z;
    i:     integer;

begin

   for i := 1 to 10 do r[i] := i+10;
   q(10, r)

end.
```

If a parameterized variable statement contains only constant parameters, the compiler may treat the variable as equivalent to a fixed type with the same parameters. This allows a container type to be used as a “template” type to create static variables:

```
program p;

type z = array of integer;

var a(10): z; ! an array of 10 integers
    b(20): z; ! an array of 20 integers

begin

end.
```

This is in fact the preferred paradigm of Pascaline, that array sizes are not part of a type, but rather a parameter of the variable has that size of array.

If the parameterized variable expression contains local variables, it is an error, since such variables are, by definition, undefined at the start of the block. Note that this does not apply to parameters, which are initialized externally to the procedure or function that contains the parameterized variable statement.

Parameterized variables allow the specification of fixed types from abstract types. This allows both the use of types for general classes of types, as well as moving the sizes of arrays out of type descriptions.

## 6.15 Extended write/writeln statements

Pascaline extends the meaning of field width parameters in ISO 7185 6.9.3.1 to include negative or zero field widths. The appearance of a negative value as a field width has the same effect as the absolute value of the field width, except that the resulting character translation is set left justified within its field. i.e., the spaces that are specified to pad each output format to the field width are output after the contents of the field, not before.

```
program p(output);

var i: integer;

begin

   write(i:-10)

end.
```

Would output the integer **i** in 10 spaces at least, with any padding to the right side of the number.

The behavior of a fielded write for integers is equivalent to:

```
program p(output);

procedure WriteInteger(var f: text; i: integer; fl: integer);

var c: integer;
    p: integer;

begin

   if fl < 0 then begin ! left justified

      ! determine the number of required characters

      c := 0;
      if i < 0 then c := c+1; ! count sign
      p := maxint; ! set maximum power
      ! count digits
      while p > 0 do
         begin if abs(i) >= p then c := c+1; p := p div 10 end;
      write(f, i:1); ! output minimum format
      if fl > c then write(f, ' ':fl-c) ! complete justification

   end else write(f, i:fl) ! output regular format

end;

begin

   WriteInteger(output, 20, -10)

end.
```

For real values in the floating point format, there is no change for left justified formatting. This occurs because there are no blanks used to justify such a number. The precision of the fraction is simply extended to fill the field. Thus:

```
program p(output);

var r: real;

begin

   write(r:-20)

end.
```

Is equivalent to:

```
program p(output);

var r: real;

begin

   write(r:20)

end.
```

There is no difference between the left justified format for a floating point representation, but there is also no error.

For a real value output in fixed point format:

```
program p(output);

var r: real;

begin

   write(r:-20:5)

end.
```

The equivalent code is:

```
program p(output);

procedure WriteRealFixed(var f: text; r: real; fl: integer;
                         fr: integer);

var c: integer;
    p: real;

begin

   if fl < 0 then begin ! left justified

      ! determine the number of required characters

      c := 2; ! minimum field includes '0.'
      if r < 0 then c := c+1; ! count sign
      p = 10.0; ! set minimum for extra digits
      ! calculate digits beyond '0.'
      while p <= abs(r) do begin c := c+1; p := p*10 end;
      write(f, r:c:fr); ! output minimum format
      ! complete justification
      if abs(f) > c then write(f, ' ':abs(fl)-(c+fr))

   end else write(f, r:fl:fr); ! output regular format

end;

begin

   WriteRealFixed(output, r, -20, 5)

end.
```

A zero field parameter has practical use only with string type and character type parameters. This is because integer and real types have minimum output format sizes. An example of use with strings is:

```
program p(output);

var i: integer;

begin

   for i := 11 downto 0 do writeln('hello there': i)

end.
```

```
Hello there
Hello ther
Hello the
Hello th
Hello t
Hello
Hello
Hell
Hel
He
H
(blank)
```

In the code:

```
program p(output);

procedure WriteSpaces(s: integer);

begin

   write(‘ ‘: s)

end;

begin

   WriteSpaces(10)

end.
```

Serves to output a given number of spaces, from 0 to N.

### 6.15.1 Radix specifiers

**write**/**writeln** statements will also accept a so called “radix specifier” on integer output statements, using one of:

| Specifier | Radix |
|---|---|
| **$** | Hexadecimal |
| **%** | Binary |
| **&** | Octal |

These are used after the value that is output:

```
program p(output);

var i: integer;

begin

   write(i$)

end.
```

The resulting output will be in the hexadecimal radix, with the same rules governing fields. Note that the radix specifier is independent of the radix specifier for integer constants:

```
program p(output);

var i: integer;

begin

   write($1234$:10)

end.
```

Is a valid program, and outputs:

```
      1234
```

### 6.15.2 Special field specifiers on write/writeln

There are two special field characters that apply to **write**/**writeln**:

| Character | Meaning |
|---|---|
| * | On string type only, means to interpret the string as padded. |
| # | On integer, means to fill the field with leading ‘0’s. |

For *, the field character replaces the field, and means that the field width is equal to the total number of characters in the string, minus the number of contiguous blank characters on the right hand side. For example:

```
program p(output);

begin

   write(‘the quick brown fox       ‘:*)

end.
```

The actual field width given the string will be 19, since that is the length of the left hand side string without the padding on the right.

Note it is an error if the operand is not a string.

For #, the field character leads the field width, which must be present. It modifies the field specification, such that instead of padding with spaces on the left hand side, the number is padded with zeros:

```
program p(output);

var i: integer;

begin

   write(42:#10)

end.
```

Would print:

```
0000000042
```

Notes:

1. The character has no meaning in a right justified number, and may give an error or be ignored.
2. It does not matter what radix specifier is used, if any.
3. It is an error if the operand is not integer or real.
4. If applied to real, it has no effect.

## 6.16 Extended read/readln statements

**read**/**readln** statements are symmetrical with their **write**/**writeln** counterparts for text files. Variable parameters can accept field specifications and radix specifiers. String variables can be read. Constant string parameters can appear, to be matched to input.

The net effect of symmetrical **read**/**readln** operations is to allow files generated by **write**/**writeln** statements to also be read by **read**/**readln** statements.

A standard ISO 7185 Pascal string variable or Pascaline string variable can appear as a **read**/**readln** parameter:

```
program p(input);

var s(10): packed array of char;

begin

   read(s);

end.
```

The equivalent code is:

```
program p(input);

var s(10): packed array of char;

procedure ReadString(var f: text; var s: string);

var i: integer;

begin

   for i := 1 to max(s) do read(f, s[i]);

end;

begin

   ReadString(input, s)

end.
```

If the end of line is encountered, it is read as a space and the read operation continues. If end of file is encountered, it is an error.

For fielded read parameters, an integer value field, which can be any expression, appears after the read parameter:

```
program p;

var f: text;
    i: integer;

begin

   read(f, i:10);

end.
```

The fielded read requires the input integer to be complete within the specified number of characters, disregarding any leading or trailing blanks.

The equivalent code to read an integer is:

```
program p;

var f: text;
    i: integer;

procedure ReadInteger(var f: text; var i: integer; fl: integer);

var s: integer;

function NextChar: char;

begin

   { if past the field, terminate with space }
   if fl = 0 then NextChar := ' '
   else NextChar := f^

end;

procedure GetChar;

begin

   get(f);
   fl := fl-1

end;

begin

   s := +1; { set sign }
   { skip leading spaces }
   while (NextChar = ' ') and not eoln(f) do GetChar;
   if not (NextChar in ['+', '-', '0'..'9']) then
      throw(InvalidIntegerFormat);
   if NextChar = '+' then GetChar
   else if NextChar = '-' then begin GetChar(f); s := -1 end;
   if not (NextChar in ['0'..'9']) then throw(InvalidIntegerFormat);
   i := 0; { clear initial value }
   while (NextChar in ['0'..'9']) do begin { parse digit }

      i := i*10+ord(NextChar)-ord('0'); { add in new digit }
      GetChar

   end;
   { make sure the rest of the field is spaces }
   while fl > 0 do begin

      if NextChar <> ' ' then throw(InvalidIntegerFormat);
```

```
      GetChar

   end

end;

begin

   ReadInteger(f, i, 10)

end.
```

Reading a character with a field is similar

```
program p;

var f: text;
    c: char;

begin

   read(f, c:10);

end.
```

The equivalent code is:

```
program p;

var f: text;
    c: char;

procedure ReadChar(var f: text; var c: char; fl: integer);

begin

   read(f, c); ! get the character
   ! make sure the rest of the field is spaces
   while fl > 0 do begin

      if f^ <> ' ' then throw(FieldNotBlank);
      get(f);
      fl := fl-1

   end

end;

begin

   ReadChar(f, c, 10)

end.
```

When reading a string, the default field is equivalent to the number of characters in the string. If a field is present, and is greater than the number of characters in the string, then the string is expected to appear with a number of leading blanks. To specify a number of trailing blanks, a negative field is specified.

In both cases, if the field is smaller than the number of characters in the string, only that number of characters will be read. There is no initialization implied for characters in the string past the number of characters in the field.

If the field is 0, it has no effect and gives no error.

```
program p;

var f: text;
    s: packed array [1..20] of char;

begin

   read(f, s:10);

end.
```

The equivalent code is:

```
program p;

var f: text;
    s: packed array [1..20] of char;

procedure ReadString(var f: text; var s: string; fl: integer);

var l: integer; { net length of string }
    i: integer;

procedure SkipSpaces(c: integer);

begin

   while c > 0 do begin

      if f^ <> ' ' then throw(FieldNotBlank);
      get(f);
      c := c-1

   end

end;

begin

   l := max(s); { find net length of string }
   if abs(fl) < l then l := fl;
   { skip leading spaces }
   if -fl > max(s) then SkipSpaces(max(s)+fl);
   for i := 1 to l do read(f, s[i]); { get string characters }
   { skip trailing spaces }
   if fl > max(s) then SkipSpaces(max(s)-fl)

end;

begin

   ReadString(f, s, 10)

end.
```

For real variables, the results are similar to that of integer:

```
program p;

var f: text;
    r: real;

begin

   read(f, r:10);

end.
```

The equivalent code is:

```
program p;

var f: text;
    r: real;

procedure ReadReal(var f: text; var r: real; fl: integer);

var i: integer;

{ find power of ten efficiently }

function pwrten(e: integer): real;

var t: real; { accumulator }
    p: real; { current power }

begin

   p := 1.0e+1; { set 1st power }
   t := 1.0; { initalize result }
   repeat

      if odd(e) then t := t*p; { if bit set, add this power }
      e := e div 2; { index next bit }
      p := sqr(p) { find next power }

   until e = 0;
   pwrten := t

end;

function NextChar: char;

begin

   { if past the field, terminate with space }
   if fl = 0 then NextChar := ' '
   else NextChar := f^

end;

procedure GetChar;

begin

   get(f);
   fl := fl-1

end;
```

```
procedure ReadInteger(var i: integer);

var s: integer;

begin

   s := +1; { set sign }
   if not (NextChar in [‘+’, ‘-‘, ‘0’..’9’]) then
      throw(InvalidRealFormat);
   if NextChar = ‘+’ then GetChar
   else if NextChar = ‘-‘ then begin GetChar(f); s := -1 end;
   if not (NextChar in [‘0’..’9’]) then throw(InvalidRealFormat);
   i := 0; { clear initial value }
   while (NextChar in [‘0’..’9’]) do begin { parse digit }

      i := i*10+ord(NextChar)-ord(‘0’); { add in new digit }
      GetChar

   end

end;

begin { ReadReal }

   { skip leading spaces }
   while (NextChar = ‘ ‘) and (fl > 0) do GetChar;
   ReadInteger(i); { read integer section }
   r := i; { convert integer to real }
   if NextChar in ['.', 'e', 'E'] then begin { it's a real }

      if NextChar = '.' then begin { decimal point }

         GetChar; { skip '.' }
         if not (NextChar in ['0'..'9']) then
            throw(InvalidRealFormat');
         p := 1.0; { initialize power }
         while NextChar in ['0'..'9'] do begin { parse digits }

            p := p/10.0; { find next scale }
            { add and scale new digit }
            r := r+(p * (ord(NextChar) - ord('0')));
            GetChar { next }

         end

      end;
      if NextChar in ['e', 'E'] then begin { exponent }
```

```
         GetChar; { skip 'e' }
         if not (NextChar in ['0'..'9', '+', '-']) then
            throw(InvalidRealFormat);
         ReadInteger(i); { get exponent }
         { find with exponent }
         if i < 0 then r := r/pwrten(i) else r := r*pwrten(i)

      end

   end;
   { make sure the rest of the field is spaces }
   while fl > 0 do begin

      if NextChar <> ' ' then throw(InvalidRealFormat);
      GetChar

   end

end;

begin

   ReadReal(f, r, 10)

end.
```

Constant strings are allowed to appear in the read/readln parameter list:

```
program p;

var f: text;
    i: integer;

begin

   read(f, 'The answer is: ', a,  ' resulting in: ', b);

end.
```

The effect of such a constant is that each character of the input is matched to the characters in the string, in turn, regardless of if the end of line is true. If the character is matched, it is read and thrown away. If the character is not matched, it is an error.

The following code shows the equivalent of such a match.

```
program p;

var f: text;
    i: integer;

procedure ReadConstant(var f: text; view s: string);

begin

   for i := 1 to max(s) do
      if f^ <> s[i] then throw(UnmatchedConstantCharacter);
   get(f)

end;

begin

   ReadConstant(f, 'The answer is: ')

end.
```

### 6.16.1 Radix conversion

When **read**/**readln** is applied to integer, the following radix conversion character can be read:

| Charcter | Radix |
|---|---|
| **$** | Hexadecimal |
| **%** | Binary |
| **&** | Octal |

When a radix character is seen before the number, it is interpreted to be in the radix specified. If no radix character is seen, the default is decimal.

Radix input formats follow numeric formats, see 6.6 “Numeric constants”.

Note that the radix character is considered part of the numbers field.

### 6.16.2 Radix specifiers

**read**/**readln** statements will also accept a so called “radix specifier” on integer input statements, using one of:

| Specifier | Radix |
|---|---|
| **$** | Hexadecimal |
| **%** | Binary |
| **&** | Octal |

This is used after the **read**/**readln** variable to be read:

```
program p;

var f: text;
    i: integer;

begin

   read(f, i$:10);

end.
```

Will result in the radix of the numeric sequence read for **i** to be assumed as hexadecimal.

Notes:

1. It is an error to use a radix specifier on any variable other than integer.
2. A radix character in the input will override any radix specifier.

### 6.16.3 Field specifier character

| Character | Meaning |
|---|---|
| * | On string type only, means to interpret the string as padded. |

Only one field specifier character can exist on a **read**/**readln**, and it replaces the field:

```
program p;

var f: text;
    s(10): packed array of char;

begin

   read(f, s:*);

end.
```

The * character means to read characters until:

1. The string is filled.
2. An eoln is encountered in the input.

If the string is completely filled, and more input characters exist, an error results. If an eoln is encountered, the **read**/**readln** stops, and the remaining part of the string specified to be read, if any, is filled with blanks.

Note it is an error to apply * to other than a string variable.

## 6.17 Type converters/restrictors

expression = simple-expression [ relational-operator simple-expression ] .

simple-expression = [ sign ] term { adding-operator term } .

term = factor { multiplying-operator factor } .

factor = variable-access | unsigned-constant | function-designator | type-identifier '(' expression ')' | set-constructor | '(' expression ')' | 'not' factor .

multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' .

adding-operator = '+' | '-' | 'or' | 'xor' .

ISO 7185 Pascal defines the function **ord** to convert any scalar or character type to integer, and defines the function **chr** to convert integer to character. Pascaline extends this system to include an enumerated types converter:

```
program p;

type a = (one, two, three);

var x: a;

begin

   x := a(1);

end.
```

In the example the result of a(1) is the enumerated type constant **two**. Transfer from an integer to an enumerated type is the only new type conversion defined in Pascaline.

A similar appearing construct is the “type restrictor”:

```
program p;

type a = 20..30;

var y: integer;

begin

   y := a(y+1);

end.
```

The assignment y := a(y+1) would seem to have no function. However, the compiler can take this as a hint that instead of promoting the expression y+1 to the full size of an integer, that the operation can be

performed in only the precision required for values within 20..30. This can enable more efficient processing of expressions on some machines.

When a type restrictor appears, the compiler can generate an error for values in the restricted value that cannot fit in the range of the target type. Type converters and restrictors never simply discard values.

The value within a type restrictor must be assignment compatible with the type specified in the type restrictor.

## 6.18 Fixed types

block = { declaration } [ 'private' declaration ] statement-part [ ';' statement-part ] .

declaration = label-declaration-part | constant-definition-part | type-definition-part | variable-declaration-part |fixed-declaration-part | procedure-declaration ';' | function-declaration ';' .

fixed-declaration-part = [ 'fixed' fixed-declaration ';' { fixed-declaration ';' } ] .

fixed-declaration = identifier ':' type-denoter '=' value-constructor.

value-constructor = array-value-constructor | record-value-constructor | constant .

array-value-constructor = 'array' value-constructor { ',' value-constructor } 'end' .

record-value-constructor = 'record' value-constructor { ',' value-constructor } 'end' .

In addition to declaring variables, Pascaline can declare a program construct that appears anywhere a variable can, but is defined completely at compile time:

```
program p;

fixed a: integer = 1;
      b: array [1..10] of integer =
            array 5, 6, 8, 2, 3, 5, 9, 1, 12, 85 end;
      c: record a: integer; b: char end = record 1, 'a' end;
      d: array [1..5] of packed array [1..5] of char = array

         'tooth',
         'trap ',
         'same ',
         'fall ',
         'radio'

      end;

begin
end.
```

The declaration of a fixed is similar to a **var** declaration, but each variable is given a "value" part that contains a value constructor. A value constructor is a series of constants or other value constructors separated by ‘,’s. As in the case of character array constant assignment to character array variables, a

constant string can be used to specify the value of a **fixed** character array, and must have the same number of characters as the definition.

Notes:

1. Fixed types can be used anywhere a variable type can, but a fixed type cannot be "threatened" in the sense of the **for** variable threat of ISO 7185 6.8.3.9.
2. The type of a fixed cannot be pointers, files, variant records, exceptions or classes.
3. The value must be assignment compatible with the variable.

A fixed declaration is logically the equivalent of a **var** declaration that is initialized at the start of it's containing block:

```
program p;

fixed a: integer = 1;
      b: array [1..10] of integer = array 5, 6, 8 end;

begin

end.
```

Is equivalent to:

```
program p;

var a: integer;
      b: array [1..3] of integer;

begin

   a := 1;
   b[1] := 5;
   b[2] := 6;
   b[3] := 8

end.
```

Except that, as mentioned, the contents of the variables **a** and **b** are protected from modification.

When a fixed type is a container, the geometry of the container is satisfied by the initializer:

```
program p;

fixed a: array of integer = array 1, 2, 3 end;

begin

end.
```

Is equivalent to:

```
program p;

fixed a: array [1..3] of integer = array 1, 2, 3 end;

begin

end.
```

This extends to any level of container:

```
program p;

fixed a: array of packed array of char = array
           'one   ',
           'two   ',
           'three '
         end;

begin

end.
```

Note that both the size of the array of strings (3) and the strings themselves (6) are set from the initializer itself.

In such a declaration, the sizes of all the containers must match, or an error will result.

## 6.19 Extended file procedures and functions

Pascaline defines several additional functions dealing with files.

### 6.19.1 assign procedure

Files that are not declared as external via the program header are normally given the property that they are anonymous and may be deleted at the end of the program. In Pascaline, such files can also be named using the function **assign**:

```
program p;

var f: text;

begin

   assign(f, 'myfile');
   reset(f);

   { statements to use file f }

   close(f)

end.
```

**assign** takes any string (defined in ISO 7185 Pascal as a packed array of characters with a starting index of 1 and any length). The format of the file name contained in the string is entirely implementation dependent, but implementations will honor, at minimum, the conventions that:

1. Leading and trailing spaces in the name are ignored, and removed if required.
2. That the characters in the set ['a'..'z', 'A'..'Z', '0'..'9', '_'] are valid in a filename.
3. The filename must begin with characters from the set ['a'..'z', 'A'..'Z','_'].

Notes:

1. These are the same conventions as used for identifiers in Pascaline.
2. This is only a subset of implementation defined filename conventions. The particular Pascaline implementation may have its own additional requirements for the formatting of filenames that are a superset of the above conventions.

The existence of a named file implies, but does not require, that the file is not deleted at the end of the program run. It further implies that applying **reset** to a named file will allow its preexisting contents to be accessed.

As in ISO 7185 Pascal, applying **reset** or **rewrite** to a file causes the file to change from the "undefined" state to "read" for **reset**, and "write" for **rewrite**. It is an error to apply **assign** to a file that is not in the undefined state, that is, to rename it while it is active.

A file must be in the undefined state to have **assign** applied to it. The effect of multiple **assign** operations applied to the same file while in the undefined state is itself undefined.

Notes:

1. It is an error on **reset** if the named file does not exist.
2. An existing file that appears in a rewrite statement will be cleared to zero length.

### 6.19.2 close procedure

To allow the processing of multiple files by name, the procedure **close** exists:

```
program p;

fixed names: array 5 of packed array 10 of char = array

       'myfile    ',
       'thisfile  ',
       'thatfile  ',
       'nextfile  ',
       'lastfile  '

    end;

var c: char;
    i: integer;

begin

   for i := 1 to 5 do begin

      assign(f, names[i]);
      reset(f);
      while not eof(f) do read(f, c);
      close(f)

   end

end.
```

**close** causes bond between the file variable and the named file itself to be broken, and the file state to be returned back to undefined. In this state, it can have **assign** applied again. In this way, a sequence of named files can be processed. **close** implies that if the file is anonymous, it is deleted, and that the file returns to being anonymous unless assigned again.

Because in Pascaline, a named file is implied to persist beyond the end of a program, it is also implied that each file that exists in a read or write state has a **close** performed on it automatically either when the program ends, or when the file variable falls out of activation.

Notes:

1. 1. It is an error to apply **close** to an undefined file.

### 6.19.3 length function

ISO 7185 Pascal defines files as a series of elements which can be examined in turn. Pascaline defines the standard function **length** to give the number of elements within a file:

```
program p(output);

var f: file of integer;

begin

   assign(f, ‘myfile’);
   writeln('The length of the file is: ', length(f));
   close(f)

end.
```

The function **length** is equivalent to:

```
program p(output);

type fi: file of integer;

var f: fi;

function length(var f: fi): lcardinal;

var l, r, p: integer;;

begin

   l := 0; { clear length }
   r := 0; { clear remainder }
   { find remaining elements }
   while not eof(f) do begin get(f); r := r+1 end;
   { rewind and count the total elements }
   reset(f);
   while not eof(f) do begin get(f); l := l+1 end;
   p := l-r; { find position from length – remainder }
   { find original position }
   reset(f);
   while p > 0 do begin get(f); p := p-1 end;

   length := l { return length }

end;

begin

   assign(f, 'myfile');
   reset(f);
   writeln('Length of myfile: ', length(f))
   close(f)

end.
```

The **length** function cannot be applied to a file of type text, because the length of line endings is implementation defined, so the length of a **text** file is implementation defined as well. It also cannot be applied to **packed** files.

Notes:

1. 1. It is an error to apply **length** to an undefined file.

### 6.19.4 location function

Each element of a Pascal file can be enumerated. In Pascaline, the elements of a file that is not of type **text** are numbered from 1 to N, where N is the last element of the file. The current location within the file is found with the built in function **location**:

```
program p(output);
var f: file of integer;
begin
   writeln('The location in the file is: ', location(f));
end.
```

The function **location** is equivalent to:

```
program p;

type fi: file of integer;

var f: fi;

function location(var f: fi): lcardinal;

var l, r, p, c: integer;

begin

   l := 0; { clear length }
   r := 0; { clear remainder }
   { find remaining elements }
   while not eof(f) do begin get(f); r := r+1 end;
   { rewind and count the total elements }
   reset(f);
   while not eof(f) do begin get(f); l := l+1 end;
   p := l-r; { find position from length – remainder }
   { find original position }
   reset(f);
   c := p; { set count from position }
   while c > 0 do begin get(f); c := c-1 end;

   location := p { return location }

end;

begin

   assign(f, 'myfile');
   reset(f);
   writeln('Location of myfile: ', location(f))
   close(f)

end.
```

Notes:

1. 1. It is an error to apply **location** to an undefined file.
2. 2. The function **location** cannot be applied to **text** files nor **packed** files.

### 6.19.5 position procedure

The current location within a file can be set by the built in procedure **position**:

```
program p;

var f: file of integer;

begin

   assign(f, 'myfile');
   reset(f);
   position(f, 10);

   { statements to use f }

   close(f)


end.
```

The procedure **position** is equivalent to:

```
program p;

type fi: file of integer;

var f: fi;

procedure position(var f: fi; p: integer);

begin

   reset(f);
   while p > 1 do begin get(f); p := p-1 end

end;

begin

   assign(f, 'myfile');
   reset(f);
   position(f, 10);

   { statements to use f }

   close(f)

end.
```

**position** will work with any element from 1 to n+1, where n is the last element of the file. **position** allows the location to be set one beyond the length of the file so that the file can be extended at the end.

Notes:

1. **position** cannot be applied to a **text** file or a **packed** file.
2. It is an error to apply **position** to an undefined file.

### 6.19.6 update procedure

In ISO 7185 Pascal, there is no way to update a preexisting file with new data. In Pascaline the procedure **update** can be used instead of **rewrite**:

```
program p;

var f: file of integer;

begin

   assign(f, 'myfile');
   update(f);

   { statements to use f }

end.
```

The procedure **update** for integers is equivalent to:

```
program p;

type fi: file of integer;

procedure update(var f: fi);

var cpy: fi;
    i:   integer;

begin

   ! copy previous contents of file

   reset(f);
   rewrite(cpy);
   while not eof(f) do begin read(f, i); write(cpy, i) end;

   ! change modes and copy back to original file

   rewrite(f);
   reset(cpy);
   while not eof(cpy) do begin read(cpy, i); write(fi, i) end

end;

var f: fi;

begin

   assign(f, 'myfile');
   update(f);

   ! statements to use f

end.
```

The file that **update** is applied to must exist.

As for **rewrite**, **update** places the file in write mode. However, **update** does not clear the previous contents of the file. **update** sets the location within the file at 1, just as **reset** does. If the common operation of updating a file at the end is wanted, the following code does this:

```
program p;

var f: file of integer;

begin

   assign(f, 'myfile');
   update(f);
   position(f, length(f)+1);

   ! statements to use f

end.
```

### 6.19.7 append procedure

**update** only works with non-text files. The built in procedure **append** performs the same action as above, but works with both text and non-text files:

```
program p(output);

var f: text;

begin

   assign(f, 'myfile');
   append(f);
   writeln(f, 'hi there');

end.
```

The procedure **append** for **text** files is equivalent to:

```
program p;

procedure append(var f: text);

var cpy: text;
    c:   char;

begin

   ! copy previous contents of file

   reset(f);
   rewrite(cpy);
   while not eof(f) do begin

      if eoln(f) then begin readln(f); writeln(cpy) end
      else begin read(f, c); write(cpy, c) end

   end;

   ! change modes and copy back to original file

   rewrite(f);
   reset(cpy);
   while not eof(cpy) do begin

      if eoln(cpy) then begin readln(cpy); writeln(f) end
      else begin read(cpy, c); write(f, c) end

   end

end;

var f: text;

begin

   assign(f, 'myfile');
   append(f);

   ! statements to use f

end.
```

The file that **append** is applied to must exist.

### 6.19.8 exists function

To determine if a file by name exists in the system, the function **exists** can be used:

```
program p(output);

var f: text;

begin

   if exists('thisfile') then begin

      assign(f, 'thisfile');
      reset(f)

   end

end.
```

The **exists** function takes a variable reference to a Pascaline string, and returns a boolean value that is **true** if the file exists.

The **exists** function can be used to determine beforehand if a **reset** will yield an error because of a non-existent file.

### 6.19.9 delete procedure

To delete a file by name, the **delete** procedure is used:

```
program p(output);

begin

   delete('oldfile')

end.
```

**delete** accepts a string, and causes the file to be removed from the system. It is an error if the file does not exist.

### 6.19.10 change procedure

To change the name of an existing file, the **change** procedure is used:

```
program p(output);

begin

   change('newname', 'oldname')

end.
```

The file with the name of the second parameter is changed to be the first parameter. Both parameters are strings. It is an error if the second parameter does not exist in the system, and also if the first parameter exists before the **change** call. It is also an error if any differences exist in the names

besides just their principal names. This rule forbids, for example, the changing of the location or system parameters of the file using the **change** call.

## 6.20 Added program header standard bindings

Besides the ISO 7185 Pascal textfile external bindings of **input** and **output**, there are three new external bindings defined by Pascaline:

`error`

This file can be used to output error messages or warnings to. It may be aliased to **output** by an implementation. The purpose of the **error** file is to provide a output file that will always go to the user's console, and cannot be redefined as routed to another display device.

`list`

This file is used to output for what is assumed to be printout from any attached hard copy device. The implementation may alias this file to the **output** file if such a device is not available, or if hard copy is not desired.

`command`

The **command** file is a text line or series of text lines that deliver instructions to the running program. Most commonly, it carries a single line containing the remaining text of the command line that activated the running program, after the command name itself is discarded. However, it can contain any instruction line, and can even be directed to be an entire file containing such commands.

The exact format of the commands and parameters contained in file are implementation dependent. If the implementation has no concept of a command line, the file can be assigned to another file, or may appear empty.

Since on many systems a command line exists as a line of text in memory, not as part of a file system, it is understood that the implementation will simulate the access to such a line as a file, if that is required.

In the Pascaline specification, implementers are reminded of the requirements of ISO 7185 6.10 "Programs" wherein it is an error if a program parameter is acted on and cannot complete that action. This clause of ISO 7185 Pascal specifically forbids ignoring the program parameters.

Further, in Pascaline it is strongly suggested that the program parameters which are common text files be connected, or can be caused to connect, to external files by binding to command file names. (See Annex C).

## 6.21 Redeclaration of forwarded procedures and functions

In Pascaline, the parameters of a forwarded procedure or function can be repeated. As in ISO 7185 Pascal, the names of the parameters are the same as the original names in the forwarded header. The new names, if they happen to be different in the repeated header, are ignored. However, the two parameter lists are checked for "congruency" using the same rules as procedure and function parameters in ISO 7185 6.6.3.6 “Parameter list congruity”.

```
program p(output);

procedure p(var i: integer; c: char); forward;

procedure p(var x: integer; y: char);

begin

   writeln('i: ', i, ' c: ', c)

end;

begin

   p(42, 'g')

end.
```

Repeating the parameters in a forwarded procedure or function both increases the self documenting nature of a program, by keeping the parameters close to the actual body of the procedure or function, as well as making it easier to create such headers, by simple block copy.

## 6.22 Anonymous function result

Pascaline has an alternate method to indicate a function result:

function-declaration = function-heading ';'directive | function-identification ';' function-block | function-heading ';' function-block .

function-block = block .

block = { declaration } [ 'private' { declaration } ] statement-part [ ';' statement-part ] .

compound-statement = 'begin' statement-sequence [ 'result' expression ] 'end' .

At the end of the main block of a function, a result statement can appear:

```
program p;

function y(a, b: integer): integer;

begin

   result a+b

end;
```

The result statement is an alternative to the ISO 7185 arrangement of assigning the result to the name of a function that is active. It can only be used in a function, and it can only be used at the end of the function. It can only give the result of the enclosing function (in contrast to ISO 7185 Pascal function results). It cannot be used in conjunction with a ISO 7185 Pascal assignment to the name of the

function appearing in either the block of the function whose result is to be set, nor any nested block. Thus, the following example is illegal:

```
{ illegal example }

program p;

function y(a, b: integer): integer;

procedure q;

begin

   y := 1

end;

begin

   result a+b

end;

begin
end.
```

But this example is legal:

```
program p;

function y(a, b: integer): integer;

function q: integer;

begin

   y := a+b;

   result 1

end;

begin

   a := q

end;

begin

end;
```

The assignment to the function result **y** is legal in **q**, even though **q** is using the result method, because the function **y** is using the ISO 7185 named function result method.

Unnamed function results give a clear way to indicate the function result, that obeys the single entry/single exit rule with the function result formed at the exit. They are also required in order to indicate the result of an operator overload function (See 6.26 “Operator overloads”).

## 6.23 Extended Function results

Function results can be any type, including structured:

```
program p;

type a(10): array of integer;

function y: a;

var z: a;
    i: integer;

begin

   for i = 1 to 10 do z[i] := i+10;

   y := a

end;

begin
end.
```

Such extended function results imply greater costs due to the need to copy the structure back to the caller.

Notes:

1. A function result cannot be a file, nor a structure having any part that is a file..
2. Function results cannot be container arrays.
3. The ISO 7185 rule that function results must be assignment compatible with the function result type remains.

## 6.24 Halt procedure

A standard Pascaline procedure **halt** exists:

```
program p;

begin

   if not exists('myfile') begin

      writeln('needed file does not exist');
      halt

   end

end.
```

The **halt** procedure causes the program, and all of its threads, to immediately terminate. It is equivalent to throwing the global exception.

The equivalent code for a program block is:

```
program p;

label 99;

begin

   if not exists(‘myfile’) begin

      writeln(‘needed file does not exist’);
      goto 99

   end;

   99:

end.
```

The **halt** procedure will terminate the enclosing program no matter which module it appears in.

## 6.25 Overloading of procedures and functions

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

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

attribute = 'overload' | 'static' | 'virtual' | 'override' | ‘operator’ .

The attribute **overload** can be used to specify that one procedure or function forms an overload group with another:

```
program p;

procedure x(y: integer);

begin

   { statements to operate on parameter y }

end;

overload procedure x(c: char);

begin

   { statements to operate on parameter c }

end;
```

Any number of procedures or functions with the same name can be joined this way into an overload group. When the procedure or function is called, the compiler determines which of the procedures or functions in the overload group is to be used to satisfy the call. Thus, both of these calls are possible with the definitions above:

```
x(1);
x('c');
```

The compiler considers three things when determining which procedure or function of an overload group is meant:

1. If the called context is a procedure or function call.
2. The number of parameters.
3. The type compatibility of the parameters.

Overload groups are ***not allowed*** to be ambiguous. There is no "preference" or "priority" involved with overloads. If two overloads are ambiguous with respect to each other, then an error will result at the time of the definition of the conflicting overload.

Two calls are ambiguous with each other if they match for procedure/function status, have the same number of parameters, and the parameters are compatible between lists, using the ISO 7185 Pascal definition of compatibility according to 6.4.6 “Assignment-compatibility” (where either parameter could be assignment compatible with the other), and also if one parameter is **char** while the other is a single character string. This means, for example, that having one parameter of type **real**, and the other of type integer, means those parameters are ambiguous with each other.

When parameter lists are compared, the mode of the parameter, or its **var**, **view** or **out** value status is ignored.

It is an error if two parameter lists "converge" with different modes. Two parameter lists are said to converge at any point in the list if that parameter and all parameters to the left of it are found to be compatible using the rules above. An example of an overload declaration that is invalid by this rule is:

```
! invalid program

program p;

procedure a(i: integer; c: char; b: boolean);

begin

   { statements of first overload  of a }

end;

overload procedure a(i: integer; var c: char; b: boolean; r: real);

begin

   { statements of second overload of a }

end;

begin
end.
```

Even though the two parameter lists would be distinct because they differ in number, the lists are convergent at parameter **c**, and parameter **c** has conflicting modes, the first being value, and the second being **var**.

The purpose of the convergence rule is to relieve compilers from having to store all the parameters in a procedure or function call before determining which routine of an overload group matches the call.

Overload groups must be completed within the same declaration block. Different modules, classes, procedures and functions cannot add to each other's overload groups.

Note that the type of a function result is not relevant to determining ambiguous overloads.

## 6.26 Operator overloads

operator-declaration = operator-identifier operator-heading '[ ':' result-type] .

operator-heading = 'operator' formal-identifier [ formal-parameter-list ] .

operator-identifier = 'not' | '+' | '-' | '*' | '/' | 'div' | 'mod' | 'and' | 'or' | '< ' | '>' | '=' | '<=' | '>=' | 'in' | 'is' | ':=' .

Besides overloading procedures and functions, it is also possible to overload expression operators:

```
program p;

type string = packed array of integer;
     pstring = ^string;

operator +(a, b: string): pstring;

var p: pstring;
    i: integer;

begin

   new(p, max(a)+max(b)); ! get result string
   for i := 1 to max(a) do p^[i] := a[i];
   for i := 1 to max(b) do p^[max(a)+i] := b[i];

   result p

end;

operator :=(out a: string; b: string);

var i: integer;

begin

   if max(a) < max(b) then throw(StringToLongForDestination);
   for i := 1 to max(b) do a[i] := b[i];
   for i := 1 to max(a)-max(b) do a[max(b)+i] := ' '

end;

begin
end.
```

Where “+” and “:=” are the operators to overload. The following operator overloads are possible:

```
program p;

type x: record i, r: integer end;

operator +(a: x): x; begin result a end;
operator –(a: x): x; begin result a end;
operator not (a: x): x; begin result a end;
operator +(a, b: x): x; begin result a end;
operator -(a, b: x): x; begin result a end;
operator *(a, b: x): x; begin result a end;
operator /(a, b: x): x; begin result a end;
operator div(a, b: x): x; begin result a end;
operator mod(a, b: x): x; begin result a end;
operator and(a, b: x): x; begin result a end;
operator or(a, b: x): x; begin result a end;
operator xor(a, b: x): x; begin result a end;
operator <(a, b: x): x; begin result a end;
operator >(a, b: x): x; begin result a end;
operator =(a, b: x): x; begin result a end;
operator <=(a, b: x): x; begin result a end;
operator >=(a, b: x): x; begin result a end;
operator in(a, b: x): x; begin result a end;
operator is(a, b: x): x; begin result a end;
operator :=(out a: x; b: x); begin end;

begin
end.
```

The ‘-‘ and ‘+’ operators can be both unary and binary, and they have different priorities. The exact type of operator is indicated by the number of parameters that appear.

The parameters to an operator function can be any type, but cannot be **var** or **out** mode parameters, with the exception of the assignment operator.

The assignment operator (‘:=’) first parameter must be **out** mode, and has a result type that must match the left side of the assignment.

The results of an **operator** function cannot be established via a ISO 7185 by name function result assignment. Instead, the anonymous result form is used (see 6.22 “Anonymous function result”).

```
program p;

! redefine string assignment as paddable

var StringOverflow: exception;

operator :=(out a: string; view b: string);

var i, l: integer;

begin

   l := max(b); while (b[l] = ' ') and (l > 0) do l := l-1;
   if l > max(a) then throw(StringOverflow)
   for i := 1 to l do a[i] := b[i];
   for i := l+1 to max(a) do a[i] := ' '

end;

var a(10): string;

begin

   a := 'hi there'

end.
```

Operator overloads are allowed for **uses** modules but not **joins** modules

The existence of an operator overload does not change the priority of the operator.

Operator overloads, unlike function and procedure overloads, use a set of rules known as "overlays" to establish which operator overload, or the system definition, to invoke:

1. Within the domain or scope, the operator overloads defined cannot be ambiguous with each other. They form an overload group as do overloaded procedures and functions.
2. Operator overloads can "overlay" other operator overloads and the system operator definitions, at other, enclosing domains. The operator overloads for each enclosing domain are examined, in turn, and selected if there is a match. Otherwise the operator overload definition search is repeated with the next level of enclosing domain. If ultimately not found, then the system definition (which belongs to the block enclosing all program constructs) will be used.

The types of built in operators with the exception of **is** and **xor** appears in ISO 7185 6.7.2 "Operators".

The use of an operator overload function to introduce so-called "side effects" will result in ambiguity as to when the side effect occurs, since Pascaline does not specify expression evaluation order.

## 6.27 Static procedures and functions

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

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

attribute = 'overload' | 'static' | 'virtual' | 'override' | ‘operator’ .

Pascaline defines a **static** attribute that is used before any **function** or **procedure** to indicate that it is used statically.

```
program p;

static procedure z;

var x, y: integer;

begin

   { statements of z }

end;

static function y;

begin

   { statements of y }

end;

begin

end.
```

The **static** word-symbol is used to indicate to the compiler that the procedure or function with that attribute will not recursively call itself, nor be called by any subprocedure or subfunction.

On some processors with limited addressing resources, using static can result in smaller and faster executable programs, sometimes dramatically so. Compilers targeting more advanced processors typically ignore the **static** word-symbol, except to check that it is used properly.

In the case where the procedure or function is contained entirely within a module, and is not visible outside the module (it is within the private section), the compiler can automatically determine the **static** status of a procedure or function. However, if it is visible outside the module, the compiler must assume it may be recursively called. In this case, the **static** attribute is required to obtain the improved code.

Note the compiler may also detect that a **static** attributed procedure or function calls itself, either directly or indirectly via another procedure or function, and issue an error.

## 6.28 Relaxation of declaration order

block = { declaration } statement-part .

declaration = label-declaration-part | constant-definition-part | type-definition-part | variable-declaration-part | fixed-declaration-part | procedure-declaration ';' | function-declaration ';' .

The declaration order in ISO 7185 Pascal of labels, constants, types, variables, then procedures and functions is relaxed. These constructs can appear in any order. However, the rule remains that each identifier must be declared before its use, with the normal ISO 7185 Pascal exception for pointer declarations.

Pointer forward references must resolve within the same type-definition-part.

Note relaxing declaration order in Pascaline is a more natural match for modular compilation.

## 6.29 Exception handling

try-statement = 'try' (statement-sequence) try-end .

try-end = (except-series | except-unconditional) [ 'else' statement ] .

except-series = except-specifier { except-specifier } [ except-unconditional ] .

except-unconditional = ‘except’ statement .

except-specifier = ‘on’ exception-identifier { ‘,’ exception-identifier } ‘except’ statement .

block = { declaration } [ 'private' { declaration } ] statement-part [ ';' statement-part ] .

statement-part = compound-statement .

compound-statement = 'begin' [try-end] statement-sequence [ ‘result’ expression ] 'end' .

The most common use for interprocedure gotos is to handle error returns easily from deep nested procedures. Pascaline provides an alternative structure. The **try** statement executes one or more statements with an "exception guard":

```
program p;

var myexception: exception;

begin

   try

      { ... }
      throw(myexception)
      { ... }

   on myexception except { statement to execute on myexception }
   except { statement to be executed on any exception }
   else { statement to be executed if no exception occurs }

end.
```

If anywhere within the guarded execution the statement:

```
throw(myexception);
```

or

```
throw;
```

is executed, any active procedures or nested statements are removed, and execution resumes with the statement after either the exception clause that matches the given exception, or the general exception statement. If no exception occurs, execution continues either with the **else** statement, if specified, or the next statement after the **try** statement.

A throw procedure without a specified exception to thow gives the “general exception”.

The program:

```
program p(output);

var myexception: exception;

begin

   try

      { ... }
      throw(myexception)
      { ... }

   on myexception except writeln('The operation failed')
   else writeln('The operation succeeded')

end.
```

Is equivalent to:

```
program p(output);

label 1;

var myexception: boolean;

begin

   myexception := false;
   begin

      { ... }
      myexception := true; goto 1;
      { ... }

   end;
   1:
   if myexception then writeln('The operation failed')
   else writeln('The operation succeeded')

end.
```

The **try** statement may carry a specific list of exceptions, followed by an optional unconditional exception, or simply an unconditional exception. An unconditional exception matches any exception. A specific exception only matches the named exception.

When the general exception is thrown, the result is as if an exception was thrown that does not appear anywhere in the program. It is the "master" exception handler that always exists in any program.

**try** statements may nest. The result of executing **throw** is to return to the innermost **try** statement currently executing. If a **goto** is used to branch outside of the **try** statement, then it will be effectively removed from activation, and any **throw** statements will go to the next outer **try** statement.

If an exception is thrown, and either there is no **try** statement active, or no **try** statement matches the exception indicated, then the result is an error. The result of unhandled exceptions is as if a **try** statement exists outside of the program that catches any exceptions that are not handled within it, and handles the exception by termination.

Exceptions are represented by a special variable defined at the system level called the **exception** type:

```
module throwit;

var myexception: exception;

procedure fault;

begin

   throw(myexception)

end;

begin
end.
```

In order to throw an exception, both the thrower and the catcher must be able to see the same exception variable. The exception to this rule is the general exception, which is available program wide.

An **exception** is a predefined type. If an exception is passed as a parameter, it must be passed with **var** or **ref** mode.

The exception handler itself can throw another exception, including the general exception. However, exceptions do not stack, nor is there any state, such as handled or unhandled, associated with exceptions.

Note exceptions do not cross thread or process boundaries. A thrown exception will only return to the same thread, and only to an active outer block of that thread.

A special version of the **try** statement exists for module constructor sections (see 6.39 “Modularity”). Modules register handlers for all of the exceptions that are thrown in that module. The reason is to allow the program and other nested modules to handle the exceptions, but ultimately handle the exception in the originating module.

```
module m(output);

var FaultException: exception;

procedure doit;

begin

   throw(FaultException)

end;

begin { constructor }
end; { constructor }

begin { destructor }

   on FaultException except writeln('*** module fault occurred')

end. { destructor }
```

The constructor for the module behaves as if it performed a **try** statement at the end of the constructor. The destructor then finishes the **try** statement. Note that the **on** construct must appear immediately after the **begin** for the destructor block.

The primary purpose of a constructor/destructor exception handler is to handle exceptions defined within the module. It is possible to handle exceptions created by other modules, but it is an error to throw an exception to an exception variable whose module is not active.

Note that when an unhandled exception occurs in a separate process, that causes the entire program to terminate.

Being a variable, an exception cannot be shared across process, monitor and channel barriers. This means that monitor exceptions are self-contained and go back to the monitor itself. General exceptions go back to the process, monitor or channel that contains them. An unhandled exception always terminates the program, no matter what thread the exception occurs on.

## 6.30 Assert Procedure

The **assert** procedure generates a runtime error if its expression is not true:

```
assert-statement = 'assert' '(' expression [',' character-string]')'
```

The expression can be any expression with a boolean result type. If the expression evaluates at runtime as false, then the program faults, otherwise it continues to run.

If a string constant is present after the exception boolean expression, it will serve to "annotate" the assert. Typically it contains the reason for the exception. It may be printed as part of any error message. Alternatively, it may have no effect.

The program:

```
program p(output);

var pi: ^integer;

begin

   pi := nil;
   assert(p <> nil, 'The pointer was nil')

end.
```

Is equivalent to:

```
program p(output);

label 99;

var pi: ^integer;

begin

   pi := nil;
   if p = nil begin

      writeln('The pointer was nil');
      goto 99

   end;

   {... }

   99:

end.
```

The **assert** procedure enables placing a large number of consistency checks within a program being developed that can be automatically removed from the code by a compiler option.

In addition, an implementation can provide extended debugging information when the assert occurs, such as program location, registers, etc.

## 6.31 Extended range types

ISO 7185 Pascal defines the results of integer expressions to lie within the range -**maxint**..**maxint**. The size of an integer is by definition chosen to be the size that is most efficient and natural on the target machine. Just as Pascaline defines operations to restrict results to numbers smaller than integer, it defines the ability to find results larger than a standard integer. If larger than integer variables and operations are specified, it is assumed that some penalty in terms of time, space or both will be added.

To specify an extended range value, a subrange is specified outside of the range -**maxint**..**maxint**. For example:

```
var li: 0..maxint*2;
```

Pascaline does not define the maximum length of such extended range types.

When an extended range value appears in an expression, the range of the result will have a range according to the following rules:

1. If either type is signed, the result is a signed type.
2. If either type is extended range, the result is an extended range type.

Operations with extended types are ***not*** guaranteed to be able to hold all the values of both operands. The implementation is only required to maintain at least the range of values in integer.

In addition, the programmer can specify the exact length of result at any point in the evaluation of an expression using range specifications. The only way to achieve an exact range of values throughout a calculation is to specify it.

Pascaline predefines three new types of integer.

```
cardinal
```

Cardinal types represent the positive integers only. The range of cardinal is:

```
0..maxcrd;
```

Where **maxcrd** is a predefined constant. **maxcrd** is the maximum unsigned value of a machine word on the target machine. It could also be equal to **maxint** on an implementation that does support extended **cardinal** types.

```
linteger
```

Long integer types represent an extended, signed integer type, defined as:

```
-maxlint..maxlint;
```

Where **maxlint** is a predefined constant. **maxlint** is typically the maximum value of a "double length" result on the target machine, that is, a value formed using two machine words. It can also be equal to **maxint** on an implementation that does not support extended **linteger** types.

```
lcardinal
```

Long cardinal types represent the positive integers only. The range of **lcardinal** is defined as:

```
0..maxlcrd;
```

Where **maxlcrd** is a predefined constant. **maxlcrd** is the maximum unsigned value of a "double length" result on the target machine, that is, a value formed using two machine words. It can also be equal to **maxint** on an implementation that does not support extended **lcardinal** types.

The interactions of binary operators on all integer based types is as follows:

| TYPE A | TYPE B | RESULT TYPE |
|---|---|---|
| **integer** | **integer** | **integer** |
| **cardinal** | **cardinal** | **cardinal** |
| **linteger** | **linteger** | **linteger** |
| **lcardinal** | **lcardinal** | **lcardinal** |
| **integer** | **cardinal** | **integer** |
| **linteger** | **lcardinal** | **linteger** |
| **linteger** | **cardinal** | **linteger** |
| **integer** | **lcardinal** | **linteger** |
| **linteger** | **integer** | **linteger** |
| **lcardinal** | **cardinal** | **lcardinal** |

Unary operators simply deliver the same type as the operand.

Note the lack of defined maximum for integer representation in Pascaline can be interpreted to implement so called “N-length integers”, which can be any length up to the size of memory in the implemented machine. This is the analog of arbitrary length sets in ISO 7185 Pascal.

### 6.32 Extended real types

Pascaline defines two new real types, **sreal** and **lreal**:

```
program p;

var sr: sreal;
    lr: lreal;

begin

end.
```

Unlike integers, Pascaline cannot set ranges for real types. Also unlike integers, there is no “natural” size of a real for most machines, as in the native type that is inherent to the word size of the machine.

**sreal** defines a **real** type that smaller and has less precision than a ISO 7185 **real**. **lreal** defines a **real** type that is larger and has greater precision than a standard ISO 7185 **real**. An **sreal** has a precision that is less than or equal to **real**, and **lreal** has a precision that is greater than or equal to **real**. In practice, **sreal**s can be equivalent to **real**s, and **lreal**s can be equalent to **real**s.

The use of an **sreal** implies that the programmer wishes to save space, and perhaps execution time over the use of an ISO 7185 Pascal **real**, in exchange for less precision and total range. The use of an **lreal** implies that the programmer wishes to obtain more precision and total range than an ISO 7185 Pascal **real**, in exchange for using more space and perhaps longer execution time. The word “perhaps”

is used here because it is common to simply extend all real types to the same length on some machines and process them identically.

All **real** types are compatible with each other, and behave as **real**s in all contexts. In equations, **real**s are always “value conserving”. The result of mixing two different types of reals is the larger of the two.

The result of using different **real** types in combination is:

| Type A | Type B | Result type |
|---|---|---|
| **real** | **real** | **real** |
| **sreal** | **sreal** | **sreal** |
| **lreal** | **lreal** | **lreal** |
| **sreal** | **real** | **real** |
| **lreal** | **real** | **lreal** |
| **Sreal** | **lreal** | **lreal** |

## 6.33 Real Limit Determination

Several new system defined constants are defined to express the limits of real numbers:

| Type | Low Limit | High Limit |
|---|---|---|
| **real** | **-maxreal** | **maxreal** |
| **sreal** | **-maxsreal** | **maxsreal** |
| **lreal** | **-maxlreal** | **maxlreal** |

## 6.34 Character limit determination

Pascaline provides a constant for character set implementation limits, **maxchar**

This character constant defines the maximum value of a character in the range:

```
chr(0)..maxchar;
```

**maxchar** is of type char.

## 6.35 Matrix mathematics

One dimensional arrays of integers or reals and two dimensional arrays of integers or reals can participate in expression operations:

| Operator | Function | Operand(s) |
|---|---|---|
| **+a** | vector/matrix affirmation | Vector, Matrix, integer or real |
| **-a** | Vector/Matrix negation | Vector, Matrix, integer or real |
| **a+b** | Vector/matrix Addition | a or b vector, matrix, integer or real |
| **a-b** | Vector/matrix Subtraction | a or b vector, matrix, integer or real |
| **a*b** | Vector/matrix Multiplication | a or b vector, matrix, integer or real |

```
program p;

type ai: array of integer;

var ai1(10), ai2(10): ai;

begin

   ai1 := ai1+ai2;
   ai1 := ai1+1

end.
```

Is equivalent to:

```
program p;

type ai: array of integer;

var ai1(10), ai2(10): ai;
    i:        integer;

begin

   for i := 1 to 10 do ai1[i] := ai1[i]+ai2[i];
   for i := 1 to 10 do ai1[i] := ai1[i]+1

end.
```

For a matrix case:

```
program p;

type ai: array of array of integer;

var ai1(10, 10), ai2(10, 10): ai;

begin

   ai1 := ai1+ai2;
   ai1 := ai1+1

end.
```

Is equivalent to:

```
program p;

type ai: array of array of integer;

var ai1(10, 10), ai2(10, 10): ai;
    x, y:                     integer;

begin

   for x := 1 to 10
       for y := 1 to 10 do ai1[x, y] := ai1[x, y]+ai2[x, y];
   for x := 1 to 10
       for y := 1 to 10 do ai1[x, y] := 1;

end.
```

Any single dimension array of integers or reals with an integer starting index of 1 qualifies as a vector. Any two dimensional array of integers or reals with both integer starting indexes of 1 and equal major and minor dimensions qualifies as a matrix.

For two vector or matrix operands to be compatible with each other, they must be of the same type, or aliases of that type, or one or more must be a container that is compatible with the other. The result is the same type as the operands.

One of the operands of +, -, or * can be an integer or real when the other side is a vector or a matrix. The result of this is a vector or matrix with the given operation performed on each element of the matrix using the single real or integer. However, a single real cannot be used with a integer matrix, as this would imply conversion of real to integer.

```
program p;

type ai: array of integer;

var ai1(10): ai;

begin

   ai1 := ai1*2

end.
```

Is equivalent to:

```
program p;

type ai: array of integer;

var ai1(10): ai;
    i:       integer;

begin

   for i := 1 to 10 do ai1[i] := ai1[i]*2

end.
```

And

```
program p;

type ai: array of integer;

var ai1(10, 10): ai;

begin

   ai1 := ai1*2

end.
```

Is equivalent to:

```
program p;

type ai: array of integer;

var ai1(10, 10):  ai;
    x, y: integer;

begin

   for x := 1 to 10
      for y := 1 to 10 do ai1[x, y] := ai1[x, y]*2

end.
```

Vectors or matrices can also be assigned as a whole:

```
program p;

type ai: array of integer;

var ai1(10): ai;

begin

   ai1 := 1

end.
```

Is equivalent to:

```
program p;

type ai: array of integer;

var ai1(10): ai;
    i:   integer;

begin

   for i := 1 to 10 do ai1[i] := 1

end.
```

And

```
program p;

type ai: array of integer;

var ai1(10, 10): ai;

begin

   ai1 := 1

end.
```

Is equivalent to:

```
program p;

type ai: array of integer;

var ai1(10, 10):  ai;
    x, y: integer;

begin

   for x := 1 to 10
      for y := 1 to 10 do ai1[x, y] := 1

end.
```

Such a vector or matrix assignment causes the expression to be assigned to each of the elements of the vector or matrix. The assignment compatibility rules are the same as non-vector or matrix assignment. If individual elements of the vector or matrix are assignment compatible with the expression, then the entire vector or matrix is also assignment compatible.

Vector or matrix expression operators do not perform a function that could not be added to Pascaline as a library function. Their primary reason for being built into the Pascaline language is that such operations can often be translated to efficient operations directly in the target hardware.

## 6.36 Saturated math operators

Many processors now directly support so called “saturated math” in single and vector operations. Saturated math is useful when manipulating real world analog values. Normally, the operators *, + and - can overflow. For example:

```
a := maxint+1;
```

Is typically an overflow operation. Saturation math operators do not overflow, but rather leave overflowed values at their maximum or minimum.

The saturation operators are:

| Operator | Meaning | Operands |
|---|---|---|
| /+ | Saturated add | Integer, vector of integer, matrix of integer |
| /- | Saturated subtract | Integer, vector of integer, matrix of integer |
| /* | Saturated multiply | Integer, vector of integer, matrix of integer |

So a typical saturated operation could be:

```
program p;

var a, b: integer;

begin

   a := 42;
   b := 12;
   a := a /* b

end;
```

There is no standard value for either the positive maximum or minimum saturated value. The correct test for the minimum or maximum is:

```
a >= maxint;
```

For the maximum saturated value and:

```
a <= -maxint;
```

For the minimum saturated value.

As for the normal *, + and - operators, the saturated operators can participate in vector and matrix operations:

```
program p;

var a(10), b(10):         array of integer;
    c(10, 10), d(10, 10): array of array of integer;

begin

   a := 1; ! initialize entire vector a to 1s
   c := 42; ! initialize entire matrix c to 42s
   a := a /* 2; ! multiply each vector element a by 2
   c := c /* 10; ! multiply each matrix element c by 10
   a := a /+ b; ! add vectors a and b
   c := c /- b ! subtract matrix b from matrix c

end;
```

## 6.37 Properties

property-identifier = qualified-identifier .

property-declaration = attribute property-identifier ‘:’ type-denoter ‘;’ property-block ‘;’

property-block = directive | read-block ‘;’ write-block ‘;’

read-block = block .

write-block = block .

A property is declared object that appears as a variable, but has the reading and writing of its contents completely defined by the program that contains it:

```
program p;

var sum: integer;

property myval: integer;

begin { read definition }

   myval := sum

end;

begin { write definition }

   sum := sum+myval

end;

begin

   sum := 0;
   myval := 10;
   myval := 5;
   writeln(‘myval is ‘, myval)

end.
```

The property declaration is followed by two blocks, the first being the read definition block, and the second being the write definition block. During the read and write definition blocks, the identifier that is the same as the name of the property carries the value that is written or read, and acts as an ordinary variable within the definition block that can be read or written. Outside of the property definition, the identifier is defined by the actions in the read and write definition.

At the end of the read definition, the content of the property identifier variable is used to satisfy the read request.

At the start of the write definition, the content of the property identifier variable is set from the write request.

Thus in the example, two write requests are made to **myval** which sums them starting with zero, and the final read request returns the resulting sum, 15.

Properties are a step beyond operator overloading, which can define the calculation of values for entire classes of variables with a given type. Properties can set the exact behavior of individual objects, even if they all have the same type.

Notes:

1. The property can be any type, including structured types, with the exception that it cannot be a file or be a structure containing a file, or an object or structure containing an object, because a file or object cannot be the target of an assignment (see 6.41 Classes).
2. Each of the read and write blocks of a property can have its own declaration section. The declaration for each of the read or write blocks applies only to that block.
3. Properties can be forwarded just as procedures and functions can.
4. Properties can be subject to overrides (see 6.39.5 ”Overrides”).

Example of property forwarding:

```
program p;

var sum: integer;

property myval: integer; forward;

property myval: integer;

begin { read definition }

   myval := sum

end;

begin { write definition }

   sum := sum+myval

end;

begin

   sum := 0;
   myval := 10;
   myval := 5;
   writeln('myval is ', myval)

end.
```

## 6.38 Common Types

Pascaline defines several "common types" that are used across many support modules. Defining these types in the system level negates the need for each module to have to include these types.

### 6.38.1 string

```
type string = packed array of char;
```

Strings are the container version of ISO 7185 strings, which have a fixed length. Because each type of string must be declared individually, there was never a standard string type.

### 6.38.2 pstring

```
type pstring = ^string;
```

A pointer to a container string. These are very useful, and typically accepted in any library where strings are.

### 6.38.3 byte

```
type byte = 0..255;
```

### 6.38.4 abyte

```
type abyte = array of byte;
```

An array of bytes.

Used in system support libraries as a generic block of memory.

### 6.38.5 vector

```
type vector = array of integer;
```

This is the definition of vectors used in built in vector/matrix math operations.

### 6.38.6 matrix

```
type matrix = array of array of integer;
```

This is the definition of two dimentional matricies used in built in vector/matrix math operations.

## 6.39 Modularity

module = module-heading ';' module-block '.' .

module-heading = module-type identifier [ '(' module-parameter-list ')' ] .

module-type = 'program' | 'module' | 'process' | 'monitor' | ‘channel’ | 'share' .

module-block = uses-declaration-part { module-declaration } [ 'private' { module-declaration } ]
[ statement-part [ ';' statement-part ] ] .

module-declaration = declaration | class-declaration .

uses-declaration-part = [ link-specification module-name { ',' module-name } ] .

link-specification = 'uses' | 'joins' .

module-name = identifier .

ISO 7185 Pascal expresses programs as a series of nested blocks, from programs to procedures and functions. Unfortunately, this model requires compilation as a single unit.

Pascaline defines a series of modules which have the same status and level as a program block, but exist in separate files. The basic module appears as:

```
module x(input, output);

uses mymodule;

var x: integer;

private

var y: integer;

begin
end.
```

A module appears very much like a program block. It exists in a separate file. It can have header parameters. It has a main block. It also can have an "ending" block. Unlike the program block, Modules are not designed to execute for the duration of the program, but to run before the program block. A module initializes its variables, then passes control on either to other modules, or the program block. If the module has a destructor block, this will be executed after the program block completes. This allows the module to perform clean up on termination, such as closing files, etc.

Modules allow a collection of resources to be created, such as labels, constants, types, variables, fixed, procedures, and functions. These are collected as a unit to serve the program. A program block can be thought of as a special case of a module that has no destructor block.

The bonding of modules and programs into an executable unit, as well as the order in which the constructor blocks and destructor blocks are executed is implementation defined.

### 6.39.1 Module parameters

Module parameters, just a for the program module, specify definitions that are satisfied by the external system. They can be system files (see 6.20 “Added program header standard bindings”), or other variable definitions. As for program, if they are not standad external bindings, then they must be declared in the **var** section of the module.

The definition of other module types besides the program module means there can be more than one module parameter list in a program module “set”. All of the module parameters will be treated as “accessed in parallel” by all modules. For files this means that successive reads and writes to the file will be interleaved as required. For other than files, the parameters are accessed in parallel by all modules, I.E., a write to a parameter will be seen by all other modules using that header parameter.

In all cases, even in the case of modules that execute as parallel tasks, accesses to the header parameters must not cause the program to malfunction.

### 6.39.2 Including other modules

The use of a module by another module or program is specified by a **uses** or **joins** declaration:

```
uses mod1,mod2;
```

or

```
joins mod1,mod2;
```

The **uses** and **joins** specifiers must precede any other declaration within a program or module. There can be only one **uses** and one **joins** specifier per program or module, and a **joins** declaration must precede a **uses** declaration.

Both a **uses** and a **joins** specification cause the declarations contained within the specified module to be imported to the outer block declarations of the importing module or program. This importation is logically done as if the source file itself were actually read. The equivalent of this action may also be performed instead by reading a preprocessed equivalent of the target program or module.

A **uses** or **joins** specification only imports that specific module. If the imported module itself imports further modules, which it may require to complete its declarations, any such import must occur without exposing that secondary module to the original importing module. This is a so-called "incidental import", and necessary. Such imports will be completed such that their declarations are available to the requesting module, but not the original importing module. This can create the situation that the importing module receives declarations of which it cannot access the component declarations within. If the importing module must access such components, it must specify the containing module in a **uses** or **joins** specification. This prevents "zipper effect", where a importing module gets an unintended series of submodules included.

Imported and importing modules cannot contain loops or mutually referring modules. This is an error in Pascaline.

All of the declarations of a module can be exported, including constants, types, variables, fixed, procedures, functions and classes with one exception. Goto labels cannot be exported, and a goto statement cannot target a module other than its own, because the context of the target label is known only to the module that contains it. The preferred method used in Pascaline to perform intramodule error recovery is to use exceptions or overrides.

The appearance of a **uses** specification causes global identifiers within the imported module to be merged without qualification to the importing module or programs name space. The identifiers can be directly used, and an error will be flagged if there is a duplicate name between importing and imported header module.

The appearance of a **joins** specification causes the identifiers to be made available in "qualified" form. A qualification appears as follows:

qualified-identifier = module-name { ‘.’ identifier }

For example:

```
mod1.alpha
```

Qualification prevents the collision of name spaces and is the preferred method of importing modules.

Notes:

1. Without a **uses** or **joins** specification, modules and programs will not conflict even if there are one or more names that are identical between them.
2. It is an error to refer to the components of a module before it has executed its start block. This is an error that may or may not be caught at runtime.
3. It is an error to call a procedure or function that is not active. This is an error that may or may not be caught at runtime.

### 6.39.3 Private Declarations

Pascaline provides a way to create declarations in a module that are not visible by importing modules or programs with the **private** specification:

```
module x;

type alpha = array 10 of char;

private

var x: alpha;

begin
end.
```

The **private** word-symbol can appear anywhere between declarations. It indicates that all further declarations in in a module cannot be exported. The appearance of the **private** word-symbol essentially divides the outer declarations of a module into a "public" section that appears first, and a private section that follows. The declarations for a module can thus be set up as a set of public declarations that define the external interface for the module, and a set of private declarations that perform the work of defining the functionality of the module.

The public and private areas cannot be mixed or changed in order. Specifically, it is not possible to create a public definition that is built with private declarations in Pascaline (so called "opaque" types). The converse is possible, private declarations based on public declarations.

Procedures and functions can be forwarded across from the public area to the private area. It is possible to create procedures and functions that use private declarations such as variables and other procedures and functions.

```
module x;

procedure y; forward;

private

procedure y;

begin
end;

begin
end.
```

Pascaline does not enforce the structure of intermodule links, nor is this within the power of the language. The most reliable structure for modules is a tree with the program module at the top, and successive layers of service and support routines below that. This kind of structure necessarily excludes loops.

### 6.39.4 Definition vs. implementation modules

Pascaline does not define a special definition as opposed to an implementation module. An implementation module which contains all of the source code required to define the contents of the module, also serves as its definition module.

If it is required to construct a module containing just the definitions within a module, and not the implementation source, the method used is to "strip" the module to give a definition module. Stripping means to remove all definitions that are private, and to remove the contents of any public procedure or functions. This will give a module that can be used to define the interface to the module for other modules, but cannot be used to build the module. It is typically used in conjunction with modules that have already been completely processed to object code in the target processor.

Such definition modules can be manually or automatically constructed by the implementation. Definition modules can be verified by compilation with sufficient options to ignore unreferenced definitions, and lack of function result assignments.

Definition modules can also be used to represent modules written in another language, or even assembly language. This facility takes the place of the **external** directive allowed for in ISO 7185 Pascal.

### 6.39.5 Overrides

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

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

attribute = 'overload' | 'static' | 'virtual' | 'override' | ‘operator’ .

Overrides give the ability for new modules to add to or replace the functionality of older modules. This may give, for example, the ability for a new module that implements graphical operations on a printer the ability to add to an existing graphical module that implements such operations on a user terminal.

A function or procedure can be specified as capable of being overridden by higher level modules. The procedure or function must specify that capability using the **virtual** word-symbol:

```
module m;

virtual procedure x;

begin
end;

begin
end.
```

When specifying a procedure, function or property as virtual, the user should realize that there is a small cost in program run time, program space, or both to handle the requirements of a virtual procedure, function or property.

To override the procedure, function or property, the word-symbol **override** is used:

```
module m;

override procedure x;

begin
end;

begin
end.
```

In many cases, a procedure, function or property that overrides another will extend or change it by handling the new features of the procedure, function or property itself, but then send the unchanged part of the service back to the overridden procedure or function. This can be done via the **inherited** word-symbol:

```
module m;

override procedure x(q: integer);

begin

   if q = 1 then inherited x(q) else { perform here }

end;

begin
end.
```

The **inherited** word-symbol specifies that the following procedure, function or property call is to be sent back to the original overridden version.

If multiple overrides of a function, procedure or property exist, they will nest. The procedure, function or property that is actually performed is the last procedure or function overridden. The function or procedure that is performed when inherited is specified is the function or procedure that was current at the time of the override.

Only one override for the same procedure, function or property can exist in a module. The inherited version of a procedure, function or property can only be called from the procedure function or property that overrides it.

The order in which overrides are executed is the same as the order of module initializations.

Notes:

1. Override procedures, functions and properties can only exist at the outer block of a module. They may not be nested.
2. The overridden virtual procedure, function or property must be external to the defining module.
3. The virtual procedure or function and its overrider must be congruous.
4. No overloads can exist to a **virtual** or **override** procedure or function.
5. The overrider of a joined module must use a qualident for the overridden routine.

Properties can be overridden similar to procedures and functions:

```
module m;

var sum: integer;

virtual property v: integer;

begin v := sum end; ! read

begin sum := sum+v end; ! write

begin
end.

module n;

override property v: integer;

begin v := inherited sum+1 end;

begin inherited sum := v end;

begin
end.
```

Note the inherited word-symbol must be used in front of each read or write instance of the property that will be sent to the overridden property definition.

A virtual procedure, function or property need not contain useful code. A typical design paradigm is to use a declaration with no useful implementation to define an “abstract” module whose functionality will be defined by further overrider modules:

```
program graphics;

var notimplemented: exception;

virtual procedure setpixel(x, y: integer);

begin

   throw(notimplemented)

end;

virtual procedure line(x, y: integer);

begin

   throw(notimplemented)

end;

begin

end.
```

Only a monitor can override other monitors. Share modules cannot contain overrides.

### 6.39.6 Parallel modules

A module in Pascaline forms the basic structure of a program. The **program** block itself is actually a special instance of a module which has no destructor. Together, **program** modules and common modules form a group that runs the main task of the program.

#### *6.39.6.1 process module*

The **process** module appears just as a **program** module. It has only a constructor, and no destructor. It can accept header parameters. The difference between a process and a program is that the process defines a new task within the program that runs in parallel with the main task, or any other process blocks.

```
process mythread(input, output);

uses mymonitor;

var x, y;

procedure q;

begin
end;

begin

   while true do writeln('I am another thread')

end.
```

A process can run "forever", in which case it simply terminates when the main task terminates, or it can run for a time and then stop. A process stops when it exits from its constructor.

A process can have a **uses** or **joins** statement, but it cannot use just any module. To do so would be to conflict with the main task run by the program module. No other **module** can use a process module. A process can use its own globals or routines, and can use a **monitor** or **share**.

#### *6.39.6.2 Monitor module*

A monitor module appears just as normal, program callable module:

```
monitor m(output);

procedure locked; forward;

private

var x, y;

procedure locked;

begin

   x := 1

end;

begin

   { startup statements }

end;

begin

   { shutdown statements }

end.
```

Within a monitor, public procedures and function of that monitor can be called.

The monitor module is a multitask "hard" module. Each of its publicly callable procedures, functions and properties have a special locking function that is executed on entry to the routine, and unlocked on exit. The net effect is that only one task running under Pascaline is allowed within a monitor at one time. This prevents the data corruption that would otherwise occur in a multitasking system.

Monitors have other special requirements that prevent data corruption. A monitor cannot have global variable definitions. All global variables must be defined within the private section. Procedures and functions can be defined in the global section, which means that they are multitask locked procedures and functions, but to accomplish useful work, they must be forwarded into the private section.

Further, the procedures and functions in the module cannot have as any parameter or return value a pointer variable, nor a structure that contains a pointer variable.

#### *6.39.6.3 share modules*

Monitors can use other monitors, atoms, channels and streams, but cannot use a **process**, **program** or **module**. However there is a module form that can be used by any other module, all of **process**, **program**, **module** and **monitor**, known as a **share** module:

```
share mylibrary(output);

procedure x;

begin
end

.
```

**share** modules cannot have any global variables at all. Because of this, it also has no startup or shutdown code blocks. There is nothing in a **share** to set up or to shut down.

The advantage to a **share** is that it can contain a series of support routines that can be accessed by any caller. A **share** acts as a **monitor** without any global data, but does not have the locking overhead of a **monitor**.

### 6.39.7 Monitor signaling

Monitors serve as more than just a library of procedures and functions callable by any task. Because a module has state in the form of globals, it can serve as a communications method between tasks. In Pascaline, using the **monitor** or **channel** modules is how multiple tasks coordinate their actions.

Given just the definition of monitors above, it would be possible for tasks to communicate by using status routines to flag conditions between them. However, each task would have to poll, or continually call a monitor function to find out that status. This wastes computer time that multitasking is supposed to use efficiently.

Instead, Pascaline defines a special variable called a **semaphore**, and a few special routines that can be used with them, the **signal**, **signalone**, and **wait** procedures.

```
monitor queue;

procedure inqueue(b: byte); forward;
procedure outqueue(var b: byte); forward;

private

const maxque = 100; { maximum length of queue }

type queinx = 1..maxque; { pointers for queue }

var fifo:     array [queinx] of byte; { fifo for queue }
    inptr:    queinx;    { in pointer }
    outptr:   queinx;    { out pointer }
    notempty: semaphore; { not empty signal }
    notfull:  semaphore; { not full signal }

{ queue pointer iterator }

function next(i: queinx): queinx;

begin

   if i = maxque then i := 1 { queue has wrapped }
   else i := i+1; { next location }

   next := i { return result }

end;

{ place byte in queue }

procedure inqueue(b: byte);

begin

   { if full, wait until a byte clears }
   while next(inptr) = outptr do wait(notfull);

   { place input byte }
   fifo[inptr] := b;

   { set next input location }
   inptr := next(inptr);

   { signal queue is now not empty }
   signal(notempty)

end;
```

```
{ get byte from queue }

procedure outqueue(var b: byte);

begin

   { if empty, wait until a byte is available }
   while inptr = outptr do wait(notempty);

   { get output byte }
   b := fifo[outptr];

   { set next output location }
   outptr := next(outptr);

   { signal queue is now not full }
   signal(notfull)

end;

begin { constructor }

   { set input = output, and queue is empty }
   inptr = 1;
   outptr = 1

end.
```

The **wait** procedure acts just as a loop to poll if the condition represented by the semaphore has occurred, but it is implemented efficiently under Pascaline. Typically, the calling task is put to sleep until the signal occurs, then it is woken up again to continue.

When a **signal** call is made, this causes any and all tasks waiting on the semaphore to be freed to run. If the signaling task knows that only one task can actually use the signal, it can use the **signalone** procedure instead. This will only free up a single task, and leave the rest to wait for the next signal.

Semaphores give tasks an easy method to flag a given event between them. They are most commonly used to pass data between tasks. A "data provider" task would place data into global structures in the monitor, then signal to any "data consumer" tasks that the monitor has data.

Semaphores can only be used within a monitor, which is the only place they have meaning. A **wait** call would not be useful if it left the monitor lock in effect, blocking any other task from signaling to it. The **wait** call releases the lock on the monitor that calls it until it is signaled and emerges from the wait. Then it reasserts the lock and continues.

Semaphores are implemented under Pascaline as "fair", or having the property that tasks that wait for a semaphore receive the signal on a "first come-first serve" basis. However, if the **signal** procedure is used to signal multiple tasks (as opposed to a **signalone** call), it is possible that one or more of the

signaled tasks could find that the condition being waited on is no longer true, since another task has serviced it first. For this reason, **wait** is used in a loop that checks for the condition being true.

Note that **signalone** could be implemented as **signal**. This means that the wait loop method should be used even with **signalone**.

The equivalent code for **semaphore**, **wait**, **signal** and **signalone** is:

```
module signal;

private

type semaphore = record

        one: boolean; ! single/multiple signal flag
        inq: integer; ! input queue pointer
        outq: integer ! output queue pointer

      end;

var s: semaphore;

! Find next queue circular position

function next(i: integer): integer;

begin

   if i = maxint then i := 0 else i := i+1

   result i

end;

! Test for queue empty

function empty(var s: semaphore): boolean;

begin

   result s.outq = s.inq

end;

! Test for queue full

function full(var s: semaphore): boolean;

begin
```

```
   result s.outq = next(s.inq)

end;

procedure wait(var s: semaphore);

var p: integer; ! our place in queue

begin

   ! if queue is full, wait for entry
   while full(s) do escape;
   s.inq := next(s.inq); ! advance input queue
   p := s.inq; ! save that position
   while s.outq <> p do escape; ! wait for our turn
   ! if there are more in queue and not single mode, advance
   if not (s.one and not empty(s)) then s.outq := next(s.outq)

end;

procedure signal(var s: semaphore);

begin

   ! flag signal complete
   if not empty(s) then s.outq := next(s.outq);
   s.one := false ! set multiple waiters ok

end;

procedure signalone(var s: semaphore);

begin

   ! flag signal complete
   if not empty(s) then s.outq := next(s.outq);
   s.one := true ! set single waiters

end;

begin ! initializer block for signal

   ! set no signal active

   s.inq := 0;
   s.outq := 0

end.
```

The procedure **escape** exists in another module. Its only purpose is to break the monitor lock in signal, which is satisfied by exiting the module and returning. It could do more, for example it could make a system call to release the rest of its task time.

```
share other;

procedure escape;

begin

end;

.
```

The equivalence implements a fair locking scheme. Each of the waiters for a signal is placed in a first in, first out queue. For **signalone**, only the head task or first in receives the signal. For **signal**, all of the waiters in the queue receive the signal.

The difference between this equivalent implementation of **semaphore**, **wait**, **signal** and **signalone** is that the system specific implementation can arrange for only the prime entry waiting on a signal is scheduled to run. Thus, there is no polling and no wasted CPU time.

## 6.40 Channels

A **channel** is one step beyond a monitor. It does not allow any procedure or subroutine to appear as public, but instead only allows properties to be exposed publically. As with a monitor, the read and write blocks of a property are locked.

```
channel queue;

property bfifo: byte; forward;

private

const maxque = 100; ! maximum length of queue

type queinx = 1..maxque; ! pointers for queue

var fifo:     array [queinx] of byte; ! fifo for queue
    inptr:    queinx;    ! in pointer
    outptr:   queinx;    ! out pointer
    notempty: semaphore; ! not empty signal
    notfull:  semaphore; ! not full signal

! queue pointer iterator

function next(i: queinx): queinx;

begin

   if i = maxque then i := 1 ! queue has wrapped
   else i := i+1; ! next location

   result i ! return result

end;

property bfifo: byte;

! place byte in queue

begin

   { if full, wait until a byte clears
   while next(inptr) = outptr do wait(notfull);

   ! place input byte
   fifo[inptr] := bfifo;

   ! set next input location
   inptr := next(inptr);

   ! signal queue is now not empty
   signal(notempty)

end;
```

```
! get byte from queue

begin

   ! if empty, wait until a byte is available
   while inptr = outptr do wait(notempty);

   ! get output byte
   bfifo := fifo[outptr];

   ! set next output location
   outptr := next(outptr);

   ! signal queue is now not full
   signal(notfull)

end;

begin ! constructor

   ! set input = output, and queue is empty
   inptr = 1;
   outptr = 1

end.
```

Note that the channel code uses the same **signal**, **signalone** and **wait** routines as does a monitor. This is the same byte queue code as given before for a monitor, refactored for channels. The resulting channel is used as follows:

```
program p;

uses queue;

var i: integer;

begin

   for i := 1 to 10 do bfifo := i;
   for i := 1 to 10 do writeln('The numbers are: ', bfifo)

end.
```

Would print:

1 2 3 4 5 6 7 8 9 10

Because the numbers are run through the byte queue in order. Such a queue can form the basis of an intertask communication system.

Channels are a more advanced multitasking object than monitors because:

1. They don't have or need (most of) the pointer passing restrictions of a monitor, since there are no parameter lists for procedures or functions.
2. The "client" and "server" of a channel can be implemented in different threads on the same processor, different shared memory processors, wholly separate processors, and even geographically separated processors connected by a network, without the overhead and problems of remote procedure calls. The server can even be implemented in hardware.

Notes:

1. The property type may not contain a pointer, nor a structure that contains a pointer..
2. A channel can contain any number of public properties.

## 6.41 Classes

class-declaration = ('class' | 'thread'| 'atom') identifier ' [ class-parameter-list ] ;' [ 'extends' class-name ';' ] block '.' .

class-parameter-list = '(' class-parameter { ';' class-parameter } ')' .

class-parameter = [ 'view' ] identifier-list ';' type-idenifier .

class-name = qualified-identifier .

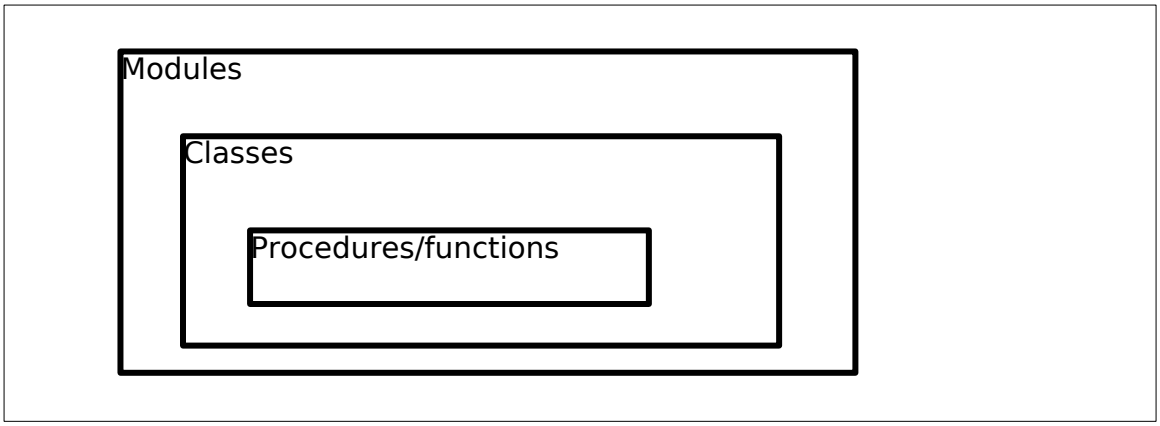
block = { declaration } [ 'private' { declaration } ] statement-part [ ';' statement-part ] .

declaration = label-declaration-part | constant-definition-part | type-definition-part | variable-declaration-part | fixed-declaration-part | procedure-declaration ';' | function-declaration ';' property-declaration | class-declaration ';' .

Modules in Pascaline provide a way to package types, variables, procedures, functions and other definitions together as a unit to be used by other parts of the program. Classes extend this further by providing a general description of a module from which the program can create any number of instances.

In Pascaline, modules are both a description of a class, and the only instance of it.

Classes fit in the program hierarchy between modules and procedures/functions:




The class is a collection of types, variables and procedures called "members", and it has per-instance data.

A class is declared by the word-symbol **class**, and it can appear anywhere in declaration:

```
uses mylib;

class myclass;

type a = char;

var c, d: integer;

procedure x;

begin

end;

private

procedure y;

begin
end;

begin

   { optional constructor }

end;

begin

   { optional destructor }

end.
```

Classes can only exist within modules.

Classes themselves define a new type.

When a procedure or function appears in a class, it is referred to as a method. A class can refer to any of the declarations in the surrounding module. The methods of a class can refer to both the variables inside the class as well as the variables in the block that surrounds it.

As a module can, a class can use the **private** word-symbol to specify declarations within the class that are private.

Note goto references cannot cross between constructor and destructor.

### 6.41.1 Qualidents

To access the members of a class, a qualident must be used just as for modules. Qualidents for classes are of the form:

qualident = [ module-name '.' ] class-name | instance-name '.' { class-name | instance-name '.' }
member-name

The first part of the qualident specifies the module name if the class exists in an external module. The name of the class then appears, and this can be repeated any number of times when classes are defined inside other classes.

Whether the class name or the instance name is used is according to the declaration kind of the member:

| Kind | Uses |
|---|---|
| **const** | class name |
| **type** | class name |
| **fixed** | class name |
| **variable** | instance name |
| **procedure or function** | instance name |

### 6.41.2 Object instantiation

A class does not create any actual variables, procedures and functions when it is defined. A class is a template for an object. To create an object from a class, it must be instantiated, which means to give the class a complete set of locations in memory. If that memory is static, the instance is static. If the memory is allocated dynamically, the instance is dynamic.

Note: objects cannot be copied from one to another, even if they are compatible. Only their members can be copied.

### 6.41.3 Static objects

To create a static instance, the class name is used as a type as follows:

```
program p;

class ci;

var a: integer;
    c: char;

.

var z: ci;

begin

   z.a := 42

end.
```

A static instance is static even if it is a local of a procedure or function, or instantiated within another class. An instance does not become dynamic unless it is allocated dynamically via **new**.

### 6.41.4 Dynamic objects

To create a dynamic instance, a reference is created to the type:

```
program p;

class ci;

var a: integer;
    c: char;

.

var z: reference to ci;

begin

   new(z);
   z.a := 42

end.
```

This creates a special variable **z** that is used to access the object that instantiates the class. The actual creation of the object is done by the standard procedures **new** and **dispose**:

The definitions for the object referenced by **z** can be accessed by a qualident:

```
z.c := 1; ! assign value to object variable
z.x; ! call object procedure x
```

A reference variable can point to any object defined with a compatible class. It can also point to no object at all:

```
z := nil;
```

Just as a pointer variable can. Further, two references can be compared, or assigned, just as pointers can:

```
program p;

class c;

begin
end.

var z, q: reference to c;

begin

   z := nil; q := z;

   if z = nil then { perform actions on nil };
   if q = z then { perform actions on q = z }

end.
```

In fact, a reference behaves just as a pointer, but without the need to specify a dereference ("^"). Anytime a qualifier (".") follows a reference, it is understood that the object referenced is being accessed.

Notes:

1. A reference is more expensive, in space and runtime, than a pointer.
2. It cannot be used to refer to the entire object referred to.
3. It can reference any base or derived class (see 6.41.8 “Inheritance”).

Just as for pointers, a reference can appear in a **with** statement:

```
program p;

class c;

begin
end.

var z: reference to c;

begin

   with z do { statements }

end.
```

This effectively opens up the scope of the object within the code.

Notes:

1. References are treated as pointers with respect to monitors. A reference cannot be exported by a monitor routine.

### 6.41.5 Classes as parameters

When a class is passed as a parameter, its dynamic or static form is irrelevant to its parameter status. The **var** or **out** mode can be used on a class as a parameter. The type used for the parameter is the class itself. Value mode class parameters cannot exist, since that would imply copying.

```
program p;

class c;

var i: integer;

begin
end.

procedure x(var y: c);

begin

   y.i := 1

end;

begin

end.
```

### 6.41.6 Classes as function results

Only references to classes can be returned as function results.

### 6.41.7 Class parameters

Classes accept parameters that are a subset of procedure and function parameters. Only value and **view** parameters are accepted in a class parameter list (with the exception of file parameters, which must always be **var** under ISO 7185 Pascal rules, or **var** parameters under Pascaline rules).

```
program p(output)

class stringc(size: integer);

var sdata: ^string; ! string data buffer
    len:   integer; ! current length

operator := (out d: string; view s: string);

var i: integer;

begin

   if max(s) > size then begin ! string is larger than our buffer

      dispose(sdata);
      new(data, max(s));
      size := max(s)

   end;
   for i := 1 to max(s) do data[i] := s[i];
   len := max(s)

end;

procedure print;

begin

   for i := 1 to len do write(data[i])

end;

begin ! constructor for string

   new(data, size) ! allocate string data
   len := 0 ! clear string

end;

begin ! destructor for string

   dispose(data) ! release space for string data

end.

var mystring(100): stringc;

begin
```

```
   mystring := 'hi there';
   mystring.print

end.
```

Class parameters are special in that they are specified when the object formed from the class template is instantiated. The class parameters are evaluated, and kept as long as the object exists. The constructor, the destructor, and all of the methods of the class can access the class parameters.

```
program p

class filelist(size: integer);

var filearray: array of text; ! array of text files

procedure open(i: integer);

begin

   rewrite(filearray[i])

end;

begin ! constructor

   new(filearray, size) ! allocate file array

end;

begin ! destructor

   for i := 1 to size do close(filearray[i]);
   dispose(filearray) ! release space for file data

end.

var files(100): filelist;
    i:          integer;

begin

   for i := 1 to 100 do files.open(i);
   for i := 1 to 100 do write(files.filearray[i], i)

end.
```

Note that the actual parameter list for the class appears where it is instantiated. For static instances, this is the variable declaration for it. For dynamic instances, it is the **new** statement that creates the object.

Typically, class parameters perform a “geometry change” within the object. That is what makes a class parameter different from just applying a method to it after its constructor executes. A method could not specify the creation of the parameterized array **filelist** above. Class parameters have many uses, and be used to specify any class configuration parameter.

The parameters for the class appear after any parameters for array template geometries. This would occur if the object was part of a container array:

```
program p

class filelist(size: integer);

var filearray: array of text; ! array of text files

procedure open(i: integer);

begin

   rewrite(filearray[i])

end;

begin ! constructor

   new(filearray, size) ! allocate file array

end;

begin ! destructor

   for i := 1 to size do close(filearray[i]);
   dispose(filearray) ! release space for string data

end.

var files(10, 100): array of filelist;
    i, j:           integer;

begin

   for j := 1 to 10 do begin

      for i := 1 to 100 do files[j].open(i);
      for i := 1 to 100 do write(files[j].filearray[i], i)

   end

end.
```

The dynamic equivalent of this would be:

```
program p

class filelist(size: integer);

var filearray: array of text; ! array of text files

procedure open(i: integer);

begin

   rewrite(filearray[i])

end;

begin ! constructor

   new(filearray, size) ! allocate file array

end;

begin ! destructor

   for i := 1 to size do close(filearray[i]);
   dispose(filearray) ! release space for string data

end.

var files(10): array of reference to filelist;
    i, j:           integer;

begin

   for j := 1 to 10 do begin

      new(files[j], 100);
      for i := 1 to 100 do files[j].open(i);
      for i := 1 to 100 do write(files[j].filearray[i], i)

   end

end.
```

Note the size parameter is now given to the dynamic creation of the files list object.

The main difference in this version is that the components of the files array can be any object compatible with filelist (see 6.41.8 “Inheritance”).

Class objects can also be in arrays formed by containers. Since containers can contain classes, but classes cannot have static containers, the parameters for the container are always to the left of any constructor parameters. Thus:

```
program p;

class a(i: integer);

begin ! constructor

end;

begin ! destructor

end.

class b(c: char);

begin ! constructor

end;

begin ! destructor

end.

var x(5, 20, 10): array of array of a;
    y(3, 12, 10, ‘a’): array of array of b;

begin

end.
```

Where x is a 5 by 20 matrix of objects of class a, and y is a 3 by 12 matrix of objects of class b.

### 6.41.8 Inheritance

class-declaration = ('class' | ‘thread’ | ‘atom’ | ‘stream’) identifier ' [ class-parameter-list ] ;' [ 'extends' class-name ';' ] block '.' .

expression = simple-expression [ relational-operator simple-expression ] .

relational-operator = '=' | '<>' | '<' | '>' | '<=' | '>=' | 'in' | 'is' .

Classes can "inherit" the definitions contained in other classes:

```
program p;

class x;

begin

end.

class y;

extends x;

begin

end.

begin { program constructor }

end.
```

The **extends** word-symbol is followed by the class that is inherited.

The meaning of this is that the new class receives all of the definitions present in the inherited class, and can then add its own. Objects have the special property that all references to a class are compatible with any class that is built by inheriting definitions from its reference class. Because class references can be freely assigned between compatible classes, this means that references are effectively compatible with the classes that it inherits, as well. This means that it is possible for some part of the definitions for a referenced class to be non-existent. Pascaline can detect and flag an error for an access to a missing definition in a referenced object either at compile time or runtime.

When a program needs to determine if the definitions it needs to access in an object exist can use a special expression operator:

```
program p;

class date;

var year, month, day: integer;

.

class datetime;

extends date;

var hour, minute, second: integer;

.

var mydatetime: reference to datetime;

procedure initdatetime(ref dt: mydatetime);

begin

   year := 2019;
   month := 1;
   day := 1;
   if mydatetime is time then
      with mydatetime do begin

      hour := 12;
      minute := 1;
      second := 1

   end

end.

begin

   new(mydatetime);
   initdatetime(mydatetime)

end.
```

The **is** operator is true if the referenced object either is the indicated class, or is a class that inherits from that class. In either case, it means that all of the definitions from the class defined as the base of the reference exist. In the example, the parameter **dt** always contains the fields of date, but does not always include all of the fields of datetime, so that is checked.

When a new member of a class is defined that has the same name as a member of the inherited class, the original member of the inherited class is “shadowed”, and becomes inaccessible in the derived class.

### 6.41.9 Overrides for objects

Just as static modules can, classes can override inherited procedures and functions.

```
program p;

class x;

virtual procedure y(view s: string);

begin

   writeln(s)

end;

.

class z;

extends x;

override procedure y(view s: string);

begin

   write(‘The string is: ‘); inherited y(s); writeln

end;

.

begin
end.
```

The main difference for the use of overrides with objects from the use with modules is that the instances of the original object keep their original procedures and functions, and only derived classes have the procedure or function replaced. Modules have the original procedure or function replaced for all callers.

Like modules, override procedures and functions must be in a different class than the virtual procedures and functions they override.

Just as for a module, the original definition of a procedure or function can be accessed by using the **inherited** word-symbol:

```
inherited y; { call the original definition of member procedure y }
```

Only one override can exist for the same method in an object. The inherited version of a method can only be accessed within the object where the override exists.

### 6.41.10 Self referencing

Within the code that makes up the object, its procedures, functions and start and exit blocks, referring to definitions within the object is done as for any module, dynamic or static. However, an object frequently needs to access the reference that defines access to it, as well. This is most common in the case where the object exists in a list of objects, and is done via the word-symbol **self**:

```
program p;

! General purpose list class

class list;

type ref = reference to list; ! reference to type

var next: ref; ! next item link in list

! Insert node to list at head

procedure insert(var root: ref);

begin

   next := root; ! link this to next
   root := self ! link root to this

end;

! Index next entry in list

procedure iterate;

begin

   self := next ! go next entry

end;

! Remove node

procedure remove(var root: ref);

var lp: ref;

begin

   if self = root then root := root.next ! gap from top
   else begin

      lp := root; ! index top of list
      while lp.next <> self do lp.iterate; ! find last entry
      lp.next := next ! gap from list

   end;
   next := nil ! clear link
```

```
end;

begin ! constructor

   next := nil ! clear next link

end.

! list of integer data values

class dlist;

extends list; ! based on list class

type ref = reference to dlist; ! reference to type

var data: integer; ! data value

! Find a list entry by value

function find(value: integer): ref;

var lp: ref;

begin

   lp := self; ! index here to search forward
   if lp <> nil do begin ! the whole list is not nil

      while (lp.data <> value) and (lp.next <> nil) do lp.iterate;
      if lp.data <> value then lp := nil ! not found

   end;

   result lp ! resulting in found value or nil

end;

begin ! constructor

   data := 0 ! clear data value

end.

var mylist, lp: dlist.ref; ! our data list

procedure newvalue(value: integer);

begin
```

```
   new(lp);
   data := value;
   lp.insert(mylist)

end;

begin

   ! place some test values in a list

   newvalue(123);
   newvalue(42);
   newvalue(1);

   ! Find an entry

   lp := mylist.find(42);

   ! And remove it

   lp.remove(root);
   dispose(lp);

   ! print remaining list entries

   writeln('The list contains:');
   lp := mylist;
   while not (lp = nil) do begin

      writeln('Value: ', lp.data);
      lp.iterate

   end

end.
```

**self** stands in for the reference that the object being operated on is based from.

### 6.41.11 Constructors and destructors

If a constructor or constructor and destructor exists for a class, they are executed when the object is created by **new**, or at the start of the object's defining block, and when the object is disposed of by **dispose**, or at the end of the object's defining block. The constructor is executed on a **new**, and the destructor is executed on **dispose**. Just as for a module, constructors give an opportunity to set up and tear down an object.

When an object has inherited classes, the derived class must specifically call the base class constructor. The derived class cannot access the members of the base class until its constructor is called:

```
program p;

class x(i: integer);

.

class y(c: char);

extends x;

begin

   x(ord(c));

   { statements accessing members of x }

end.

var yi('a'): y;
    xi(42): x;

begin

end.
```

The constructor of a derived class is the only place such an explicit constructor call can take place. The base class is called specifically by name with a constructor parameter list. Outside of the derived class constructor, the base class parameter list is used as normal for classes.

The explicit call of base class constructors gives two important abilities:

1. It allows the derived class to set up the parameters it needs for the base class.
2. It allows the derived class to have a different list of parameters than the base class.

Only the constructors of base classes are called explicitly. The destructor is called automatically, in the reverse order of construction, when the object is disposed or removed from scope. When an object is torn down, the derived class has the ability to deactivate itself before the class it is based on is deactivated.

Note the Pascaline processor may or may not treat accesses to members of the base object before initializations as an error.

### 6.41.12 Operator overloads in classes

When an operator overload occurs in a class, it can use the general form to refer to the specific object at hand:

```
program p;

class complex;

var r, i: real;

operator +(var lv: complex; var rv: complex);

begin

   lv.r := lv.r+rv.r; ! add real part
   lv.i := lv.i+rv.i ! add imaginary part

end;

begin ! constructor for complex
end.

begin ! program constructor
end.
```

Pascaline allows a short form where the left side of the operator always is the class of the execution context:

```
program p;

class complex;

var r, i: real;

operator +(ref rv: complex);

begin

   r := r+rv.r; ! add real part
   i := i+rv.i ! add imaginary part

end;

.

begin ! program constructor
end.
```

That is, the leftmost operand of the operator function parameters is left off. The result is that the left side of the operator is performed in the context of the object on the left. The result is just as if the present class was declared as the first parameter of the operator function. For unary functions + and -, no parameter list is specified.

Operator overloads for classes require that one or more of the operands be of the type for the containing class. This means that operator overloads for the class must pertain to that class.

As for module based operator overloads, the inherited keyword can be used to access any overlaid operator definition.

### 6.41.13 Derivation vs. composition

Pascaline supports new classes formed from old classes by either ***inheritence*** or ***composition***. Inheritance is a class that has another class as its base class. Composition is a class that includes an instance of the class:

```
program p;

class base;

var i: integer;

.

class composition;

var bi: base;

.

var ci: composition;

begin

   ci.bi.i := 10

end.
```

Building classes by composition is a powerful technique, and sometimes is more appropriate than building classes by derivation. Multiple inheritance in Pascaline is not possible using inherited classes, but it can be done with composition.

### 6.41.14 Parallel classes

Just as modules have different versions for parallel tasking, classes come in different forms to enable tasking. The following classes are the functional equivalents of the module types:

| Module type | Class type |
|---|---|
| **process** | **task** |
| **monitor** | **atom** |
| **channel** | **stream** |
| **share** | <none> |

#### 6.41.14.1 task class

A **task** class has no visible members at all. Its constructor starts and is executed within a new thread. All members must be private:

```
program p;

uses mymonitor;

task lighttask;

private

var a: integer;

begin ! constructor for lighttask

   a := getdata;
   putdata(a)

end.

begin
end.
```

A **process** module is limited to creating only a single parallel thread of execution within the program. The **task** class can create an unlimited number of threads.

#### 6.41.14.2 atom class

**task** classes and process modules can use either monitor modules, or can use a special form of class known as the **atom**:

```
program p;

atom queue;

procedure inqueue(d: byte); forward;
procedure outqueue(var b: byte); forward;

private

const maxque = 100; { maximum length of queue }

type queinx = 1..maxque; { pointers for queue }

var ! fifo for queue
    fifo:     array [queinx] of byte;
    inptr:    queinx;    { in pointer }
    outptr:   queinx;    { out pointer }
    notempty: semaphore; { not empty signal }
    notfull:  semaphore; { not full signal }

{ queue pointer iterator }

function next(i: queinx): queinx;

begin

   if i = maxque then i := 1 { queue has wrapped }
   else i := i+1; { next location }

   next := i { return result }

end;

{ place data in queue }

procedure inqueue(b: byte);

begin

   { if full, wait until a byte clears }
   while next(inptr) = outptr do wait(notfull);

   { place input byte }
   fifo[inptr] := b;

   { set next input location }
   inptr := next(inptr);

   { signal queue is now not empty }
```

```
   signal(notempty)

end;

{ get data from queue }

procedure outqueue(var b: byte);

begin

   { if empty, wait until a byte is available }
   while inptr = outptr do wait(notempty);

   { get output byte }
   b := fifo[outptr];

   { set next output location }
   outptr := next(outptr);

   { signal queue is now not full }
   signal(notfull)

end;

begin { constructor }

   { set input = output, and queue is empty }
   inptr = 1;
   outptr = 1

end.

var bq: queue;
    b:  byte;
    i:  integer;

begin ! program constructor

   for i := 1 to 10 do bq.inqueue(i);
   for i := 1 to 10 do begin

      bq.outqueue(b);
      writeln('The value is: ', b)

   end

end.
```

This is the same queue example as given for monitors, but refactored for atoms.

An **atom** cannot have variables in its public section. It is “sealed” and must perform all of its actions via methods or properties.

Each of the methods or properties of an **atom** acquires a lock specific to the atom before performing its work. Semaphores and signals are available to an atom. Thus, it can serve as a go-between for task classes, processes, and the main program.

#### *6.41.14.3 stream class*

A **stream** goes further still than an **atom**, and allows only properties to be present in its interface section.

```
program p;

stream queue;

property bfifo: byte; forward;

private

const maxque = 100; ! maximum length of queue

type queinx = 1..maxque; ! pointers for queue

var fifo:     array [queinx] of byte; ! fifo for queue
    inptr:    queinx;    ! in pointer
    outptr:   queinx;    ! out pointer
    notempty: semaphore; ! not empty signal
    notfull:  semaphore; ! not full signal

! queue pointer iterator

function next(i: queinx): queinx;

begin

   if i = maxque then i := 1 ! queue has wrapped
   else i := i+1; ! next location

   result i ! return result

end;

property bfifo: byte;

! place byte in queue

begin

   { if full, wait until a byte clears
   while next(inptr) = outptr do wait(notfull);

   ! place input byte
   fifo[inptr] := bfifo;

   ! set next input location
   inptr := next(inptr);

   ! signal queue is now not empty
   signal(notempty)
```

```
end;

! get byte from queue

begin

   ! if empty, wait until a byte is available
   while inptr = outptr do wait(notempty);

   ! get output byte
   bfifo := fifo[outptr];

   ! set next output location
   outptr := next(outptr);

   ! signal queue is now not full
   signal(notfull)

end;

begin ! constructor

   ! set input = output, and queue is empty
   inptr = 1;
   outptr = 1

end.

var bs: queue;
    i:  integer;

begin ! program constructor

   for i := 1 to 10 do bs.bfifo := i;
   for i := 1 to 10 do writeln('The value is: ', bs.bfifo:1)

end.
```

And the output would be:

1 2 3 4 5 6 7 8 9 10

This is the same code as the channel example, but refactored for streams.

#### 6.41.14.4 Class equivalent for share

There is no class based version of a **share** module. Since a share module contains no variables, there would be no point to an instantiation of such a class.

# A Annex: Collected syntax

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

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

adding-operator = '+' | '-' | 'or' | 'xor' .

apostrophe-image = "'" .

array-type = 'array' [ dimension-specifier ] 'of' component-type .

array-value-constructor = 'array' value-constructor { ',' value-constructor } 'end' .

array-variable = variable-access .

assignment-statement = ( variable-access | function-identifier ) ':=' expression .

attribute = 'overload' | 'static' | 'virtual' | 'override' .

base-type = ordinal-type .

binary-digit = '0' | '1' | '_' .

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

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

block = { declaration } [ 'private' { declaration } ] statement-part [ ';' statement-part ] .

Boolean-expression = expression .

buffer-variable = file-variable "'" .

case-constant = constant .

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

case-constant-list = case-constant-range { ',' case-constant-range } .

case-index = expression .

case-list-element = case-list-statement | case-list-default .

case-list-default = 'else' statement .

case-list-statement = case-constant-list ':' statement .

case-statement = 'case' case-index 'of' case-list-element { ';' case-list-element } [ ';' ] 'end' .

character-string = '"' string-element { string-element } '"' .

class-declaration = ('class' ‘thread’ | ‘atom’ | ‘stream’) identifier ' [ class-parameter-list ] ;' [ 'extends' class-name ';' ] block '.' .

class-name = qualified-identifier .

class-parameter-list = ‘(‘ class-parameter { ‘;’ class-parameter } ‘)’ .

class-parameter = [ ‘view’ ] identifier-list ‘;’ type-idenifier .

component-type = type-denoter .

component-variable = indexed-variable | field-designator .

compound-statement = 'begin' [try-end] statement-sequence [ ‘result’ expression ] 'end' .

conditional-statement = if-statement | case-statement | try-statement .

constant = [ sign ] constant-term { adding-operator constant-term } .

constant-definition = identifier '=' constant .

constant-définition-part = [ 'const' constant-definition ';' { constant-definition ';' } ] .

constant-factor = '(' constant ')' | 'not' constant-factor | character-string | constant-identifier | unsigned-number .

constant-identifier = qualified-identifier .

constant-member-designator = constant [ '... ' constant ] .

constant-set-constructor = '[' [ constant-member-designator { ', ' constant-member-designator } ] ']' .

constant-term = constant-factor { multiplying-operator constant-factor } .

control-variable = entire-variable .

decimal-integer = digit-sequence .

declaration = label-declaration-part | constant-definition-part | type-definition-part | variable-declaration-part | fixed-declaration-part | procedure-declaration ';' | function-declaration ';' | operator-declaration ‘;’ | property-declaration | class-declaration ‘;’ .

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ‘_’ .

digit-sequence = digit { digit } .

dimension-specifier = index-specifier | range-specifier .

directive = letter { letter | digit } .

domain-type = type-identifier .

else-part = 'else' statement .

empty-statement = .

entire-variable = variable-identifier | 'self' .

enumerated-type = '(' identifier-list ')' .

except-series = except-specifier { except-specifier } [ except-unconditional ] .

except-specifier = ‘on’ exception-identifier { ‘,’ exception-identifier } ‘except’ statement .

except-unconditional = ‘except’ statement .

expression = simple-expression [ [‘inherited’] relational-operator simple-expression ] .

factor = variable-access | unsigned-constant | function-designator | type-identifier '(' expression ')' | set-constructor | '(' expression ')' | 'not' factor .

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

field-designator-identifier = qualified-identifier .

field-identifier = qualified-identifier .

field-list = [ ( fixed-part [ ';' variant-part ] | variant-part ) [ ';' ] ] .

field-specifier = field-identifier .

file-type = 'file' 'of' component-type .

file-variable = variable-access .

final-value = expression .

fixed-declaration = identifier ':' type-denoter '=' value-constructor .

fixed-declaration-part = [ 'fixed' fixed-declaration ';' { fixed-declaration ';' } ] .

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

for-statement = 'for' control-variable ':=' initial-value ( 'to' | 'downto' ) final-value 'do' statement .

formal-identifier = identifier .

formal-parameter-list = '(' formal-parameter-section { ';' formal-parameter-section } ')' .

formal-parameter-section = value-parameter-specification | variable-parameter-specification | view-parameter-specification | out-parameter-specification | reference-parameter-specification | procedural-parameter-specification | functional-parameter-specification .

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

function-identification = attribute 'function' identifier .

function-identifier = qualified-identifier .

functional-parameter-specification = function-heading .

goto-statement = 'goto' label .

hex-digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | '_' .

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

hex-integer = '$' hex-digit-sequence .

identified-variable = pointer-variable '"' .

identifier = letter | '_' { letter | digit | '_' } .

identifier-list = identifier { ',' identifier } .

if-statement = 'if' Boolean-expression 'then' statement [ else-part ] .

index-expression = expression .

index-specifier = '[' index-type { ',' index-type } ']' .

index-type = ordinal-type .

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

initial-value = expression.

initialized-identifier = identifier [ ‘(‘ expression { ‘,’ expression ‘)’ } ] .

instance-type = 'instance' 'of' class-name .

label = digit-sequence | identifier .

label-declaration-part = [ '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' | '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'.

link-specification = 'uses' | 'joins' .

member-designator = expression [ '..' expression ] .

module = module-heading ';' module-block '.' .

module-block = uses-declaration-part { module-declaration } [ 'private' { module-declaration } ]
[ statement-part [ ';' statement-part ] ].

module-declaration = declaration .

module-heading = module-type identifier [ '(' module-parameter-list ')' ] .

module-name = identifier .

module-parameter-list = identifier-list .

module-type = 'program' | 'module' | 'process' | 'monitor' | 'share' .

multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' .

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

new-pointer-type = '^' domain-type .

new-structured-type = [ 'packed' ] unpacked-structured-type .

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

octal-digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '_' .

octal-digit-sequence = octal-digit { octal-digit } .

octal-integer = '&' octal-digit-sequence .

operator-declaration = operator-identifier operator-heading ‘[ ‘:’ result-type] .

operator-heading = attribute 'procedure' formal-identifier [ formal-parameter-list ] .

operator-identifier = ‘not’ | ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘div’ | ‘mod’ | ‘and’ | ‘or’ | ‘< ‘ | ‘>’ | ‘=’ | ‘<=’ | ‘>=’ | ‘in’ | ‘is’ | ‘:=’ .

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

ordinal-type-identifier = type-identifier .

out-parameter-specification = 'out' identifier-list ':' type-identifier .

pointer-type = new-pointer-type | pointer-type-identifier .

pointer-type-identifier = type-identifier .

pointer-variable = variable-access .

procedural-parameter-specification = procedure-heading .

procedure-and-function-declaration-part = { ( procedure-declaration | function-declaration ) ';' } .

procedure-block = block .

procedure-declaration = procedure-heading ';' directive | procedure-identification ';' procedure-block | procedure-heading ';' procedure-block .

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

procedure-identification = attribute 'procedure' identifier .

procedure-identifier = qualified-identifier .

procedure-statement = procedure-identifier ( [ actual-parameter-list ] | read-parameter-list | readln-parameter-list | write-parameter-list | writeln-parameter-list ) .

property-identifier = qualified-identifier .

property-declaration = attribute property-identifier ‘:’ type-denoter ‘;’ read-block ‘;’ write-block ‘;’

qualified-identifier = identifier { '.' identifier } .

range-specifier = unsigned-integer { ',' unsigned-integer } .

read-block = block .

read-element = variable-access | character-string .

read-parameter-list = '(' [ file-variable ',' ] read-element { ',' read-element } ')' .

readln-parameter-list = [ '(' ( file-variable | read-element ) { ',' read-element } ')' ] .

real-type-identifier = type-identifier .

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

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

record-value-constructor = 'record' value-constructor { ',' value-constructor } 'end' .

record-variable = variable-access .

record-variable-list = record-variable { ',' record-variable } .

reference-parameter-specification = ‘ref’ variable-declaration ‘;’ { variable-declaration ‘;’ } ] .

reference-type = 'reference' 'to' class-name .

relational-operator = '=' | '<>' | '<' | '>' | '<=' | '>=' | 'in' | 'is' .

repeat-statement = 'repeat' statement-sequence 'until' Boolean-expression .

repetitive-statement = repeat-statement | while-statement | for-statement .

result-type = simple-type-identifier | pointer-type-identifier .

scale-factor = [ sign ] digit-sequence .

set-constructor = '[' [ member-designator { ',' member-designator } ] ']' .

set-type = 'set' 'of' base-type .

sign = '+' | '-' .

signed-integer = [ sign ] unsigned-integer .

signed-number = signed-integer | signed-real .

signed-real = [ sign ] unsigned-real .

simple-expression = [ sign ] term { [‘inherited’] adding-operator term } .

simple-statement = empty-statement | assignment-statement | procedure-statement | goto-statement .

simple-type = ordinal-type | real-type-identifier .

simple-type-identifier = type-identifier .

special-symbol = '+' | '-' | '*' | '/' | '=' | '<' | '>' | '[' | ']' | '.' | ',' | ':' | ';' | '^' | '(' | ')' | '<>' | '<=' | '>=' | ':=' | '..' | '(.' | '.)' | '@' | word-symbol .

statement = [ label ' :' ] ( simple-statement | structured-statement ) .

statement-part = compound-statement .

statement-sequence = statement { ';' statement } .

string-character = one-of-a-set-of-implementation-defined-characters .

string-element = apostrophe-image | string-character .

structured-statement = compound-statement | conditional-statement | repetitive-statement | with-statement .

structured-type = new-structured-type | structured-type-identifier .

structured-type-identifier = type-identifier .

subrange-type = constant '..' constant .

tag-field = identifier .

tag-type = ordinal-type-identifier .

term = factor { [ ‘inherited’] multiplying-operator factor }.

try-end = (except-series | except-unconditional) [ 'else' statement ] .

try-statement = 'try' (statement-sequence)] try-end .

type-definition = identifier '=' type-denoter .

type-definition-part = [ 'type' type-definition ' ;' { type-definition ';' } ] .

type-denoter = type-identifier | new-type .

type-identifier = qualified-identifier .

unpacked-structured-type = array-type | record-type | set-type | file-type | reference-type | instance-type .

unsigned-constant = unsigned-number | character-string | constant-identifier | 'nil' .

unsigned-integer = decimal-integer | hex-integer | octal-integer | binary-integer .

unsigned-number = unsigned-integer | unsigned-real .

unsigned-real = digit-sequence '.' fractional-part [ 'e' scale-factor ] | digit-sequence 'e' scale-factor .

uses-declaration-part = [ link-specification module-name { ',' module-name } ] .

value-constructor = array-value-constructor | record-value-constructor | constant .

value-parameter-specification = identifier-list ':' type-identifier .

variable-access = entire-variable | component-variable | identified-variable | buffer-variable .

variable-declaration = variable-identifier-list ':' type-denoter .

variable-declaration-identifier = identifier [ ‘(‘ expression { ‘,’ expression ‘)’ } ] .

variable-declaration-identifier-list = variable-declaration-identifier { ‘,’ initialized-identifier } .

variable-declaration-part = [ 'var' variable-declaration ';' { variable-declaration ';' } ] .

variable-identifier = qualified-identifier .

variable-identifier-list = initialized-identifier { ‘,’ initialized-identifier } .

variable-parameter-specification = 'var' identifier-list ':' type-identifier .

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

variant-part = 'case' variant-selector 'of' variant { ';' variant } .

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

view-parameter-specification = 'view' identifier-list ':' type-identifier .

while-statement = 'while' Boolean-expression 'do' statement .

with-statement = 'with' record-variable-list 'do' statement .

word-symbol = 'and' | 'array' | 'begin' | 'case' | 'const' | 'div' | 'do' | 'downto' | 'else' | 'end' | 'file' | 'for' | 'function' | 'goto' | 'if' | 'in' | 'label' | 'mod' | 'nil' | 'not' | 'of' | 'or' | 'packed' | 'procedure' | 'program' | 'record' | 'repeat' | 'set' | 'then' | 'to' | 'type' | 'until' | 'var' | 'while' | 'with' | 'forward' | 'module' | 'uses' | 'private' | 'external' | 'view' | 'fixed' | 'process' | 'monitor' | 'share' | 'class' | 'is' | 'overload' | 'override' | 'reference | 'joins' | 'static' | 'inherited' | 'self' | 'virtual' | 'try' | 'except' | 'extends' | 'on' | 'result' | 'operator' | 'instance' | ‘start’ .

write-block = block .

write-parameter = expression [ ':' expression [ ':' expression ] ] .

write-parameter-list = '(' [ file-variable ',' ] write-parameter { ',' write-parameter } ')' .

writeln-parameter-list = [ '(' ( file-variable | write-parameter ) { ',' write-parameter } ')' ] .

Notes:

1. The syntax BNFs do not dictate semantics. A program that is perfectly valid according to the above syntax may be invalid semantically.

## B Annex: Standard exceptions

Normally, exceptions are either declared by the program or imported from external modules. However, several exceptions exist that are generated by the support libraries used to implement the base Pascaline language. These exceptions are all declared at the system level in Pascaline, and any program can handle such exceptions.

The following are a list of standard exceptions. If an implementation does not use a particular exception, it is recommended that the exception name be provided, even if the exception will never be thrown. This will insure software compatibility for applications that reference the exceptions here.

| Name | Meaning |
|---|---|
| **ValueOutOfRange** | Value out of range |
| **ArrayLengthMatch** | Array lengths do not match |
| **CaseValueNotFound** | Case value not found |
| **ZeroDivide** | Zero divide |
| **InvalidOperand** | Invalid Operand |
| **NilPointerDereference** | Nil pointer dereference |
| **RealOverflow** | Real overflow. |
| **RealUnderflow** | Real underflow. |
| **RealProcessingFault** | Real processing fault. |
| **TagValueNotActive** | Tag value for enclosing variant not active. |
| **TooManyFiles** | Too many files active. |
| **FileIsOpen** | File is open, should be closed. |
| **FileAlreadyNamed** | Attempt to name a file already named. |
| **FileNotOpen** | File is closed, should be open. |
| **FileModeIncorrect** | File in incorrect mode. |
| **InvalidFieldSpecification** | Invalid field specification. |
| **InvalidRealNumber** | Invalid real number. |
| **InvalidFractionSpecification** | Invalid fraction specification; |
| **InvalidIntegerFormat** | Invalid integer format |
| **IntegerValueOverflow** | Integer value overflow. |
| **InvalidRealFormat** | Invalid real format. |
| **EndOfFile** | End of file encountered. |
| **InvalidFilePosition** | Invalid file position specified. |
| **FilenameTooLong** | The filename passed to the system was too long. |
| **FileOpenFail** | Unable to open a file. |
| **FileSizeFail** | Unable to find size of file. |
| **FileCloseFail** | Unable to close a file. |
| **FileReadFail** | Unable to read a file. |
| **FileWriteFail** | Unable to write a file. |
| **FilePositionFail** | File position fails. |
| **FileDeleteFail** | File delete fails. |
| **FileNameChangeFail** | File name change fails. |

| | |
|---|---|
| **SpaceAllocateFail** | Dynamic space allocation fails. |
| **SpaceReleaseFail** | Dynamic space release fails. |
| **SpaceAllocateNegative** | Attempt to allocate negative space. |
| **CannotPerformSpecial** | Cannot perform operation on special file. |
| **CommandLineTooLong** | Command line too long. |
| **ReadPastEOF** | Read attempted past file EOF. |
| **FileTransferLengthZero** | Attempted file transfer of zero length. |
| **FileSizeTooLarge** | File size is too large to return. |
| **FilenameEmpty** | Filename is empty. |
| **CannotOpenStandard** | Cannot open standard I/O file. |
| **TooManyTemporaryFiles** | Too many temporary files in use. |
| **InputBufferOverflow** | Input buffer overflows. |
| **TooManyThreads** | Maximum number of threads exceeded. |
| **CannotStartThread** | Cannot start new thread |
| **InvalidThreadHandle** | Invalid thread handle |
| **CannotStopThread** | Cannot stop thread |
| **TooManyIntertaskLocks** | Maximum number of interthread locks exceeded. |
| **InvalidLockHandle** | Invalid interthread lock handle |
| **LockSequenceFail** | Invalid intertask lock sequence detected |
| **TooManySignals** | Too many intertask signals active at once |
| **CannotCreateSignal** | Cannot create new intertask signal |
| **InvalidSignalHandle** | Invalid intertask signal handle |
| **CannotDeleteSignal** | Cannot delete intertask signal |
| **CannotSendSignal** | Cannot send intertask signal |
| **WaitForSignalFail** | Failure waiting for intertask signal |

If an exception occurs that is not one of the above, it goes to the general exception.

If a program handles a system exception, but does not correct the problem, the program may simply fail later in the run, perhaps with a different exception. It is not advised to simply capture and then ignore exceptions.

The implementation typically connects the list of standard exceptions to equivalent exceptions in its implementation libraries. In this case, the complete set of implementation specific exceptions can be utilized by directly referencing the exceptions in the implementation libraries.

## C Annex: Undefined program parameter binding

To further the idea of program header parameter binding to external entities, this annex gives a suggested method of such bindings.

Program parameters are divided into three classes:

1. Predefined parameters, such as **input**, **output**, **error**, **list** and **command**.
2. Undefined text files.
3. Other than text type parameters.

Pascaline defines bindings for the first category, and this annex details a standard methodology for binding the second. The third category is left to the implementation, but the requirement exists for there to be an error if these parameters cannot be bound correctly (see ISO 7185 6.10 "Programs").

The suggested binding method for text file program parameters with names other than the predefined names is called "command parameter binding". For each such program parameter, reading from left to right in the program header, each file is connected to a parameter of the command used to activate the program as follows:

1. A filename is parsed from the command file.
2. The filename parsed is **assign**ed to the parameter file.

The equivalent code to do this is:

```
procedure get_command_filename(var f: text);

var fn: packed array maxlin of char;

begin

   for i := 1 to 100 do fn := ' '; { clear filename }
   { skip leading spaces }
   while (command^ = ' ') and not eoln(command) do get(command);
   { get file characters }
   i := 1; { set first character position }
   while (command^ <> ' ') and not eoln(command) do begin

      fn[i] := command^;
      get(command);
      i := i+1

   end;
   assign(f, fn) { set filename }

end;
```

Where **maxlin** is the maximum string length, and assuming that **command** appears as a header parameter.

The result is that the next **reset** or **rewrite** applied to the file will cause the file, with the name specified in the command file, to be activated. Note that this method does not allow the file to be closed, then reopened, since the **close** procedure is defined as removing the name of the file from the file variable.

The side effect of parsing the actual file name from the command file is that the filename is removed from the **command** file, since the current read position of the file is skipped.

The point at which the above binding of the program parameters to command file filenames is before the module that contains the program parameters is activated. This means that further reads from the command file (if it appears in the header) read data that appears after any text file names have been parsed and skipped.

# D Annex: Character sets

Like ISO 7185 Pascal, Pascaline has no standard character set representation. Individual characters can be any size, and the mapping of character numerical values to character glyphs or figures is entirely system dependent.

The character set used must provide all of the characters in the word-symbols and special symbols in Pascaline, with the addition of a space.

This annex describes two different character sets that are commonly used with Pascaline, the ISO 8859-1 and ISO 10646 sets. For the purposes of this annex, it is assumed that users of alternative ISO 8859 code pages will be migrating to the ISO 10646 standard (see below).

## D.1 ISO 8859-1 Character Set Encodings

Pascaline source code itself uses only the lower page (128) ISO 8859-1 codes, which are common to all ISO 8859 pages, and only uses a subset of those.

The ISO 8859-1 characters require a set of 256 elements (or 32 bytes) to be able to represent the declaration:

```
set of char;
```

## D.2 ISO 10646 Character Set Encodings

The ISO 10646 or Unicode character standard specifies up to 32 bits of character values.

| Variant | Bits | Set of char requirement |
|---|---|---|
| **UTF-8** | 8 | 32 bytes |
| **UTF-16** | 16 | 8192 bytes |
| **UTF-32** | 32 | 536870912 bytes |

The exact format of a character under the Unicode system is system dependent. It commonly varies according to if the machine implemented on arranges its bytes from most significant to least (so called "big endian"), or from least significant to most (so called "little endian").

Unicode uses the first 0 to 127 character values to represent the ASCII (or ISO 646) character set. It uses the first 0 to 255 values to represent the 8859-1 or latin-1 character set. Unicode also includes all of the other parts of the ISO 8859 code pages in higher value codes.

## D.3 Unicode text file I/O

When considering Unicode formats, it is necessary to consider both the internal memory format, and the form as written and read from Pascal text files. In the case of Unicode, they are not necessarily the same.

There are several in-file possible text representations for Unicode. The most common are:

| Name | Bits | Endian mode |
|---|---|---|
| **UTF-8** | 8 | None |
| **UTF-16** | 16 | Big or little endian |
| **UTF-32** | 32 | Big or little endian |

Although Unicode represents up to 32 bits of character code information, it is possible to represent all of these codes in either the UTF-8 or UTF-16 formats due to their use of escapes.

When big and little endian modes are considered, there are a total of possible combinations of 5 different unique formats.

There is no runtime method defined in standard Pascaline to set or switch between input and output text formats. Such selection is usually performed by a compile time change that may include special libraries, compiler options, or some combination.

The "Pascaline normal" or suggested default format for text file I/O, is UTF-8. UTF-8 has the following special advantages:

1. It is compact. It represents the most common characters in the fewest number of bits.
2. It is downward compatible with ASCII or ISO 646. When a text file consists entirely of ASCII characters, the file will be identical to an ASCII file.
3. It does not have any endian mode.

To be fair, UTF-8 also has disadvantages:

1. When representing files whose contents are mainly outside the 0-255 code range, it may actually increase the net size of the text file.
2. It moves the ISO 8859-1 128-255 code range characters outside of the 8 bit range, making text files containing such characters possibly larger, and obfuscates the representation of such characters.

Use of UTF-8 makes it possible for Pascaline compilers that allow both Unicode and ISO 8859 support to switch character set modes and still keep the same text file format.

## D.4 Use of different character sets

Pascaline is designed to use only one character set at a time. There is no built in system to represent more than one character set type at once, nor to switch between character set types at runtime. The configuration for character set is done as a compile time option.

# E Annex: Character escapes

Pascaline, like ISO 7185 Pascal, does not specify a particular character set. However, the most common implementations of character sets, are ISO 8859-1, and Unicode with representations from ISO 10646. All of these character codes have ASCII as a subset. Not required, but suggested for standard Pascaline implementations are character escape sequences. The purpose of a character escape sequence is to allow access to non-printable characters, also known as "control characters" within these character sets. Additionally, there may be characters that, while they are printable, cannot be represented in program source without a particular font or character set loaded into the host operating system. In this case, character escapes allow access to these character by character code number.

The appearance of an escape within a character string is indicated by the ‘\’ character. Following the ‘\’ character is one of:

1. <number>
2. <control-identifier>
3. <char>

The last construct simply means that the meaning of any character, including backslash itself, can be "forced" by prefacing it with a backslash. The backslash itself must appear via a "backslash image" of ‘\’.

The <number> can be any valid character sequence code. High quality Pascaline implementations will allow '$', '&' or '%' radix marked numbers to appear after the backslash character. Example:

```
‘This is a line\$d’
```

The digits that constitute the character are limited by radix and number of digits. That is, any digit outside of the radix of the number will terminate the escape sequence. For example:

```
‘The character is: \&0018’
```

Gives an octal character escape of "01", followed by the character ‘8’, because 8 is beyond the octal radix.

Similarly, only as many digits as are needed to form a character in the current character set of the compiler are used:

| Base | ISO 8859-1 | UNICODE 16 bit | UNICODE 32 bit |
|---|---|---|---|
| **Decimal** | 3 digits | 5 digits | 10 digits |
| **Octal** | 3 digits | 6 digits | 11 digits |
| **Hexadecimal** | 2 digits | 4 digits | 8 digits |
| **Binary** | 8 digits | 16 digits | 32 digits |

So for example:

‘The character is: \$0045’

is not the character for $45, but the character value $00 followed by the characters "45". Leading zeros are not discarded in numbers.

It is possible for the escape sequence to be ambiguous. In this case, the backslash character must also be used as a terminator:

‘The number is: \$c0’

Is ambiguous if the desired string is 'The number is ', followed by the character for code 12 and ‘0’, so use instead:

‘The number is: \$0c0’

Control identifiers give easy access to control codes in the ISO 8859-1 character set. For ISO 8859-1, they are:

| Identifier | Meaning |
|---|---|
| **NUL** | Null character |
| **SOH** | Start of Header |
| **STX** | Start of Text |
| **ETX** | End of Text |
| **EOT** | End of Transmission |
| **ENQ** | Enquiry |
| **ACK** | Acknowledgment |
| **BEL** | Bell |
| **BS** | Backspace |
| **HT** | Horizontal Tab |
| **LF** | Line feed |
| **VT** | Vertical Tab |
| **FF** | Form feed |
| **CR** | Carriage return[h] |
| **SO** | Shift Out |
| **SI** | Shift In |
| **DLE** | Data Link Escape |
| **DC1,XON** | Device Control 1 |
| **DC2** | Device Control 2 |
| **DC3,XOFF** | Device Control 3 |
| **DC4** | Device Control 4 |
| **NAK** | Negative Acknowledgement |
| **SYN** | Synchronous Idle |
| **ETB** | End of Trans. Block |
| **CAN** | Cancel |
| **EM** | End of Medium |
| **SUB** | Substitute |
| **ESC** | Escape |
| **FS** | File Separator |
| **GS** | Group Separator |
| **RS** | Record Separator |
| **US** | Unit Separator |
| **DEL** | Delete |
| **PAD** | Padding Character |
| **HOP** | High Octet Preset |
| **BPH** | Break Permitted Here |
| **NBH** | No Break Here |
| **IND** | Index |
| **NEL** | Next Line |
| **SSA** | Start of Selected Area |
| **ESA** | End of Selected Area |
| **HTS** | Character Tabulation Set |
| **HTJ** | Character Tabulation with Justification |
| **VTS** | Line Tabulation Set |
| **PLD** | Partial Line Forward |
| **PLU** | Partial Line Backward |

| | |
|---|---|
| **RI** | Reverse Line Feed |
| **SS2** | Single Shift 2 |
| **SS3** | Single Shift 3 |
| **DCS** | Device Control String |
| **PU1** | Private Use 1 |
| **PU2** | Private Use 2 |
| **STS** | Set Transmit State |
| **CCH** | Cancel Character |
| **MW** | Message waiting |
| **SPA** | Start of Guarded Area |
| **EPA** | End of Guarded Area |
| **SOS** | Start of String |
| **SGCI** | Single Graphic Character Introducer |
| **SCI** | Single Character Introducer |
| **CSI** | Control Sequence Introducer |
| **ST** | String Terminator |
| **OSC** | Operating System Command |
| **PM** | Privacy Message |
| **APC** | Application Program Command |
| **NBSP** | No Break Space |

So the sequence can appear:

```
'Welcome\cr'
```

If the underlying implementation of Pascaline is not ISO 8859-1, then the control mnemonics would be different. In addition, there may be other control identifiers defined for a particular implementation or character set.

```
'Hi there\del, george'
```

The use of character escapes causes a compatibility issue with ISO 7185 Pascal. In ISO 7185 Pascal, the character string:

```
'Hi there\george'
```

Would include the backslash, whereas a Pascaline implementation would remove the backslash. For this reason, standard complying Pascaline implementations must have a flag that enables the use of character escapes that can be turned on or off to enable normal treatment of programs. Since ISO 7185 Pascal requires that any Pascaline implementation have an option that will restrict the input language to strict ISO 7185, the character escape option must also be automatically set to off by the ISO 7185 compatibility flag. The reason that the character escape option should be a separate option, and not simply tied to the ISO 7185 flag, is that it would then be impossible to have ISO 7185 complying programs work with Pascaline full language mode enabled. This would be against the rule that Pascaline is, or can be, a completely upward compatible superset of standard ISO 7185 Pascal.

# F Annex: Interpretation of packed

Although both Pascaline and ISO 7185 do not define the effect of packed, it impacts the function of programs in Pascaline, and I give a optional interpretation here.

## F.1.1 Possible modes of packing

There are many possible modes of packing. The operands being packed also may or may not appear in the order found in the program. However there are two modes of interest here:

| Packing method | Result |
|---|---|
| **None** | No packing is done |
| **Critical** | Starting with the first bit of storage, each unit of data is placed in successive bits until the data is exhausted. |

Most compilers default to “none” as a packing method, which on a byte oriented machine effectively means “pack to byte boundaries”. Critical packing, however, can be quite useful. For example:

```
program p;

type floating = packed record

                   sign: boolean;
                   exponent: 0..255;
                   fraction: 0..8_388_607

                end;

begin
end.
```

Can be a direct description of an IEEE 754 single precision floating point number, using critical packing.

Used with in-memory data structures, this kind of packing can be used to pass data to other programs and to hardware. For example, IEEE floating point formats are a common internal representation.

Used with files, this kind of packing can be used to communicate with other programs and use existing file formats.

A commonly used device with critical packing is to use single bits to "pad" the format:

```
program p;

type floating = packed record

                   a: 0..31;
                   pad1: boolean;
                   b: char

                end;

begin
end.
```

Because critical packing is expensive, there is typically a compiler option to invoke it.

### F.1.2 Effect of packing on files

It is fairly common to place a series of arbitrary data structures into a file. In many languages this is done by converting the structure to bytes before writing it to a file. With the correct file packing method, Pascaline can directly model such files without the need for type escapes.

The packing method applies to variant records:

```
program p;

type data_record_type = (none, int, ch);
     data_record = record case ty: data_record_type of
                      none: ();
                      int: (i: integer);
                      ch: (c: char);
                   end;

var f: packed file of data_record;

begin
end.
```

Since files can be read and written purely sequentially, the in-file formats can be:

byte

| | |
|---|---|
| 0 | 0 (none) |

byte

| | |
|---|---|
| 0 | 1 (int) |
| 1 | Integer |

byte

| | |
|---|---|
| 0 | 2 (char) |
| 1 | Character |

In other words, the first part of each record in the file is the tagfield, followed by the contents of that variant, which could be empty. There is no need to pad the entry in the record to make it the same size as all other records in the file.

This type of packing gives up the ability to random access records in the file, because they vary in length.

This kind of packing can be used to read and write files for use by other programs, and use existing file formats.

# G Annex: Overview of standard libraries and modularity

In the annexes that follow, several standard libraries are introduced. The introduction of standard libraries into a standard is based on the following two ideas:

1. Pascaline has sufficient extendibility that major new applications can be covered with extension modules using the standard language, and not changes to the base language.
2. A typical Pascaline program is going to have library dependencies in addition to language dependencies.

The modules in the standard set are divided into two basic types:

1. Basic extensions used by programs regardless of extra devices or capabilities available.
2. Extensions that introduce new device capabilities.

The following extension libraries will be shown in this standard:

Annex G: Services library - Directory lists, file name handling, paths, environment, program execution, date and time, internationalization.

Annex H: String library - String routines, alternate base I/O, formatting.

Annex I: Math library – Various floating point math extensions and utilities.

Annex J: Terminal library - Output to text surface, advanced input.

Annex K: Graphical library - Output to graphical surface.

Annex L: Windowing library - Management of multiple windows.

Annex M: Widget library - Buttons, lists, dialogs relevant to screen based programs.

Annex N: Sound library - Midi and wave input and output.

Annex O: Network library - Network program access.

## G.1 Basic Language Support

### G.1.1 services

**services** contains a series of operating system support extensions such as filename/path handling, directory listings, time and date, environment strings, executing other programs, and similar functions.

### G.1.2 strings

Implements common string handling routines, dynamic string management, numeric conversion to and from a string, advanced numeric formatting to and from both strings and files, string search and replace, and similar functions.

## G.2 Advanced User I/O and Presentation Management

The advanced user I/O modules are a family that implement a series of advancing levels that match advancing levels in I/O capabilities, including:

Terminal character presentation

Gives the ability to place characters on a grid for a typical fixed size character display.

Graphical presentation

Gives the ability to draw figures, and place characters with proportional fonts.

Window management

Divides the display surface into a series of independent windows.

Widget placement and management

Gives a library of common controls, layered components and dialogs.

All of the supported display modes are upward compatible, and fixed size character presentation is always available in any graphic mode. The result is that there is two possible stacking levels of presentation:

```
Terminal  -> Single fixed display -> Windowed display -> Widgets
Graphical -> Single fixed display -> Windowed display -> Widgets
```

The terminal, graphical, window management and widget libraries are a set of libraries that implement a series of screen surface operating standards in an ascending sequence of capability:

```
serial I/O - I/O of single characters with line orientation and
|            blocking  on input.
+--Terminal I/O - I/O to an X-Y fixed font surface with arbitrary
   |        positioning and event driven I/O.
   +--Graphical I/O - I/O to arbitrary pixel locations and multiple
   |            figure types.
   +-----Multiple windows - I/O to multiple windows.
   +--------Widget - Catalog of predefined widgets and dialogs
```

These layers are all forward and backward compatible, so that any lower level can be used in any higher level. For example, a ISO 7185 Pascal that only inputs and outputs in terms of characters formatted in terms of lines can be run under any level all the way up to a multiple windowing environment with widgets without source change.

The primary method for performing this is the use of the "character grid", which shows where the cells of screen characters exist in a graphical system. The character grid is only relevant to characters as a figure, and not to lines or other figures, and output to the grid is the default. A program use arbitrary character placement by explicitly specifying it, and it can turn off the automatic character wrapping features of the grid.

Similarly, all text and graphical windows are buffered in a multiple windowing system, so that they need not be aware of the windows management unless they wish to be.

Serial I/O is the name given here of the default method of ISO 7185 Pascal which goes back to its origin, and includes the **write**, **read**, **writeln**, **readln**, **page** and similar built in procedures.

Terminal I/O is so named after the terminals that were used with computers for several decades, and for text mode still in use even in windowed operating systems.

From here, two distinct branches exist, one with, and one without graphics capability. For example, it is possible to have both multiple windows, widgets and dialogs all while staying within the capability of a character I/O only terminal. Therefore, there exists both a graphics version of the windows management library and the windows library, and one of each with character only ability.

The names of the libraries, as appears in a uses or joins statement, is as follows:

| Name | Contents | Call model |
|---|---|---|
| **terminal** | Terminal I/O | Terminal |
| **graphics** | Graphical I/O | Graphical |
| **windows** | Managed windowing | Terminal and graphical |
| **widgets** | Widgets and dialogs | Terminal and graphical |

The serial level does not require an explicit library, since that is the normal I/O method specified in ISO 7185 Pascal.

The key to understanding the variation in libraries is that a **uses** or **joins** of a particular library does not necessarily lead to a completely different block of code. A system that only has graphical output modes will implement the terminal library by aliasing that to the graphical library. It is also common for a graphical or terminal mode library to include all of the windows management and widget/dialog functions in the same module. Thus, a **uses** or **joins** of the windows management libraries does not cause a specific library to linked as much as flag that the program will be using these features.

### G.2.1 Naming

The functionality of the I/O modules nest, which effectively means that each of the static classes represented by the I/O modules extend each other. Unfortunately, there is no mechanism in Pascaline for modules to extend one another. The net effect is that when a module is referenced that is a base class of a static module, that reference is aliased to the derived class. For example, references to calls in **terminal** are rerouted to **graphics**, since graphics contains all of the procedures and functions of terminal.

The aliasing of names must cover the full ability of I/O modules in Pascaline to extend other I/O modules. This means that these combinations are possible:

| Module | Extended to |
|---|---|

| | |
|---|---|
| **terminal** | **windows** |
| **terminal** | **widget** |
| **graphics** | **widget** |

The method this is carried out is implementation specific. One typical method is to utilize interface only modules (modules with the implementation stripped out). Each of the I/O modules has its own interface specification, which includes a superset of the base module. This allows each module to be fully checked against its own specification. A program that specifies **terminal** only sees **terminal** definitions, a program that specifies graphics only sees **graphics** definitions, and so forth. At the lowest level (assembly language or "linker" level), aliases are provided for the module names, and so a program compiled to the **terminal** standard can be linked to a graphics module, etc.

Note that this issue does not exist for classes, and each of the I/O modules features a series of nested classes.

## G.3 Advanced device libraries

### G.3.1 sound

**sound** gives the ability to drive a midi output, perform sequencing, and output wave files.

### G.3.2 network

**network** allows access to a network using the ISO 7185 Pascal file model.

## G.4 Classes

Many of the standard modules expose a series of classes that gives the same functionality as the modular version, using object based design. This only occurs where there is "state" that would make sense as carried with the object. For example, **terminal** contains a class that holds the I/O files used to input and output, as well as several internal states such as text colors and modes. **services** does not carry state, and so classes are not useful there. This reflects the idea in Pascaline that modules are a static version of a class.

## G.5 Library procedure and function notation

For purposes of brevity, BNF notation is used to show optional constructs, and the **view** and **overload** word-symbols have been omitted.

The construct [x] means that x is optional.

The libraries description continues the idea that program examples be compilable from the source material. However, the parts of the interface definitions have been broken into sections by the same name. For example, terminal contains several source modules named terminal, and the understanding is that these are different sections of the same interface module.

# H Annex: System Services Library

**Services** contains common operating system related tasks, including directory access, time and date, files and paths, file attributes, environment strings, the local option character, and execution of external programs.

## H.1 Filenames and Paths

A file specification is composed of a path, name and extension:

<path><name><ext>

The exact format of a file specification changes with the operating system. The routine **brknam** takes a file specification and breaks it down into its path, name and extension components. The routine **maknam** does the opposite, creating a composite name from the three components. When a file specification has no path, it means that it refers to the default path. When a file specification needs to be printed or saved in an absolute format, with the path always specified, the **fulnam** routine is used to "normalize" the file specification by filling out the complete path.

To parse file specifications from the user, the routine **filchar** returns a character set of all possible characters in a filename. This can be used to get a sequence of characters from a string or file that can constitute a file specification. Once the potential file specification has been loaded into a string, it can be checked for validity as a file specification by **validfile**, and as a path by **validpath**.

Filenames are based on the idea that there is a set of characters that may be used for filenames in a particular installation, and the characters outside that set are used to delimit between filenames. This characteristic of filenames allows a program to load the filename into a string, then check it’s proper structure using subsequent calls. The program may remove certain characters from the set of filename characters to be able to parse command lines.

A common method used to represent filenames in command lines and other text when the characters allowed in a filename are essentially unlimited is to quote the filename. Using this method, the filename appears as:

“myfile”

or

‘myfile’

This can work together with limited character set filenames by recognizing the leading quote. If both types of quotes are allowed, then the leading quote should match the trailing quote. Additionally, the program should be able to recognize a “quote image” of the forms:

| Character sequence | Result |
|---|---|
| “” | Double quote “ |
| ‘’ | Single quote ‘ |
| \” | Forced double quote “ |
| \’ | Force single quote ‘ |

A robust program would recognize all of these forms.

Note that **services** does not contain any method to parse filenames.

If a file specification contains wildcards, this can be determined by **wild**. **services** does not define what wildcards characters or specifiers are used, or their format. **wild** simply indicates that the filename contains a wildcard specification that may result in multiple files being indicated by the same file specification.

## H.2 Predefined paths

To find common objects that a program needs, three predefined paths are provided:

Program path

> Is the path that the program was executed from. This is used to find data that accompanied the program, and system wide option files. **getpgm** is used to read the path.

User Path

> This is the path for the current user's home directory. This is used to store options that only apply to the current user. **getusr** is used to read the path.

Current Path

> The default path is used to find options that apply only to the current file being worked on. Unlike the other path types, the current path can be both read and set. Setting the current path will set the default path used to finish incomplete file specifications. **getcur** is used to read the path, and **setcur** is used to set it.
>
> Note that if the system has no concept of a current path, this property is used to form full path names from default path names in **services**.

## H.3 Time and Date

Time is kept in two different formats in **services**. The first is seconds time, and the second is "clock" time. Seconds time is literally a count of the number of seconds since a fixed reference time. Clock time is a free running clock that ticks every 100 Microseconds.

Seconds time is returned by the **time** function. It is in a format called "S2000", and it is the number of seconds relative to midnight on the morning of January 1, year 2000. This means that S2000 has a negative value for years before 2000, and a positive value after that. S2000 time is dependent on the representation of an **linteger**. This results in the following limits according to bit size:

| Bits | Farthest year into the past | Farthest year into the future |
|---|---|---|
| **32** | 1932 | 2068 |
| **64** | 1608591645726 | 1608591649726 |

Note that these times don't account for leap years, which are less than a year of error in 32 bits in any case. 16 bits cannot reasonably be used to represent a time, since the seconds in a year exceed the format.

This represents a reasonable range of time for 32 bits, and by the year 2068 it is likely that the time will universally be 64 bits or better. Note that it does not matter how many bits are returned by time, so that transition will occur seamlessly. Because 64 bits is already in excess of what is needed for computer based timekeeping, it is likely that time in 64 or more bits will actually contain fewer bits, despite the **linteger** representation.

S2000 is self-relative, meaning that times relative to the year 2000 are measured by a fixed number of seconds. S2000 only matched UTC time once, exactly at midnight on the morning of January 1, year 2000. All times after that or before that or after that are increasingly diverge from UTC. In particular, this means that the current time and date will not match current UTC.

There are two types of calculations that can be applied to UTC, fixed and dynamic. The fixed calculation typically finds UTC by applying leap year corrections, then a fixed table of leap second years. The fixed calculation will fall out of accuracy from the time that the system is released. It is impossible for it to be otherwise, since UTC is based on astronomical observation.

The dynamic calculation relies on receiving a current table of corrections from the network or other communications method. This is typically the same calculation as fixed time, but with a continually updated table of leap seconds.

When an S2000 time is not available, it is customary to set it to -**maxlint**, which makes it clear that the time was not set.

The time returned by **time** is in GMT or "universal" time. To convert to local time, the function **local** is used. It takes the given GMT S2000 time, and offsets it by the local time zone offset, and by daylight savings time, and returns the adjusted local time.

The time can be placed, in character format, to a string by **times**, and written to either an output file, or the standard **output** file by **writetime**. The date can be placed, in character format, to a string by **dates**, and written to either an output file, or the standard output by **writedate**. These procedures format the time and date using the current internationalization settings of the host.

**clock** time is typically derived from a free running counter kept in the host computer, and it is represented by **lcardinal** values. It may or may not be synchronized to the values returned by time. For this reason, **clock** should be treated as self-relative and not compared to **time** in any way.

**clock** time is treated differently from **time**. Since it free runs, you must be prepared for it to "wrap", or suddenly start counting up from zero. This can be determined by if any stored time is greater, or later in time, than the current clock value. The function **elapsed** takes a reference time, and determines how much time has passed from that, including compensation for wraparound. The exact count at which the **clock** count wraps is system dependent.

The total amount of time that can be represented with **clock** is determined not only by the bit size of the **clock** return value, but also by the size of the counter the host computer maintains. All that is guaranteed is that clock will be able to keep a unique time for at least 24 hours.

The actual increment of time for each tick of the clock is determined by the host computer. If the host cannot time to 100 microsecond accuracy, then the **clock** time will increment in multiples > 1, or effectively a running approximation of the actual timer. For this reason, there may not be an exact time length between successive counts of the timer.

## H.4 Directory Structures

The **list** procedure takes a file specification, including wildcards, and returns a linked list of all of the matching directory entries:

```
{ attributes }

attribute = (atexec,  { is an executable file type }
             atarc,   { has been archived since last modification }
             atsys,   { is a system special file }
             atdir,   { is a directory special file }
             atloop); { contains heriarchy loop }

attrset = set of attribute; { attributes in a set }

{ permissions }

permission = (pmread,  { may be read }
              pmwrite, { may be written }
              pmexec,  { may be executed }
              pmdel,   { may be deleted }
              pmvis,   { may be seen in directory listings }
              pmcopy,  { may be copied }
              pmren);  { may be renamed/moved }

permset = set of permission; { permissions in a set }

{ standard directory format }

filptr = ^filrec; { pointer to file records }

filrec = record

   name:   pstring;   { name of file }
   size:   lcardinal; { size of file }
   alloc:  lcardinal; { allocation of file }
   attr:   attrset;   { attributes }
   create: linteger;  { time of creation }
   modify: linteger;  { time of last modification }
   access: linteger;  { time of last access }
   backup: linteger;  { time of last backup }
   user:   permset;   { user permissions }
   group:  permset;   { group permissions }
   other:  permset;   { other permissions }
   next:   filptr     { next entry in list }

end;
```

Each directory file entry has the name of the file, along with a series of descriptive data for the file. These are divided into attributes and permissions. An attribute is a characteristic of the file, and generally does not change. Permissions indicate what can be done with the file, and are divided into the user, group and other permissions.

Not all attributes nor all permissions are available on every operating system. If a particular permission or attribute is not implemented on a given operating system, then setting it will have no effect, and reading it will always return unset.

The size of the file is its size in bytes. The allocation is the total space it occupies on the storage medium, which may be different from its size for several reasons. The blocking may be such that the size is rounded up to the nearest block. The operating system may have the ability to reserve space for the file beyond what it is currently using, or may not release space back to the free space pool if the file is truncated.

If the allocation is not known, it will be the same as the file size.

The times of interesting events in the files life are available, in "S2000" format (already discussed). If a particular time is not available, then it is set to -**maxlint**.

The file structure is a collection of items that may be implemented on any given operating system. The way to prevent the need to decide what is and what is not implemented on a particular system is to focus on what is essential for all systems. For example, the size of a file is usually present, as well as the last modification time. The last modification time can be used to determine when to back up files, by comparing it to the date of the backup copy of the same file, or to the modification date of the archive containing the file. Similarly, a "make" style program can determine when to remake a file by looking at the modification time.

## H.5 File Attributes and Permissions

The attributes of a file can be set by name with the **setatr** command. It takes a set of attributes and the filename, and sets all the given attributes on the file. The routine **resatr** resets attributes. The user permissions for a file are set and reset by **setuper** and **resuper**. The group permissions are set and reset by **setgper** and **resgper**. The other permissions for a file are set and reset by **setoper** and **resoper**.

## H.6 Environment Strings

The environment is a collection of strings that is kept by the executive, and passed to programs when they are started. Each string has a name and a value, both of which are arbitrary strings. An environment string can be retrieved by name by **getenv**, and set by **setenv**. The entire environment string set can be retrieved at one time by the **allenv** routine, which uses a linked list to represent the environment strings:

```
{ environment strings }

envptr = ^envrec; { pointer to environment record }

envrec = packed record

   name: pstring; { name of string }
   data: pstring; { data in string }
   next: envptr { next entry in list }

end;
```

The list is terminated by **nil.**

The standard format for environment string names is the same for Pascaline identifiers, i.e., a character in the sequence 'A'..'Z', 'a'..'z', '_', followed by any number of characters in the sequence 'A'..'Z', 'a'..'z', '_', '0'..'9'. The data in the string can be a series of any valid characters.

The reason for retrieving the entire environment is to pass it on to other programs in an **exec** statement.

Although most available operating systems implement the environment string concept, it has fallen out of favor in modern programs. Placing the needed configuration strings for a particular program into a place where the executive will use it both requires special calls, and places individual program data into a pool where it can be deleted or corrupted.

Some current systems use a repository concept where program data is kept in a central tree structured database. This is not covered in **services**, and would be covered in another library.

A better method is to use a file containing the configuration information for the program in its startup or program directory, then optionally another version in the user directory, and finally one in the current directory. This allows options that affect all runs on the current machine to be kept in one file, while the options for a particular user are kept in another file, and lastly the options in use in the current project in the current directory. This system has the advantage that it addresses concerns in a multiuser operating system, and allows each program to maintain its own startup data.

To this end, the startup or program directory is usually write protected.

For systems that do not implement an environment string capability, implementations of Pascaline commonly keep a file in the user path area that contains the strings. This is then used just for that program, that is, the set of environment strings are kept just for the accessing program.

## H.7 Executing Other Programs

An external program can be executed by **exec**, which takes the command line for the program, including the program name, and all of its parameters and other options on the same line after one or more spaces:

cmd parameter parameter... parameter

This is passed as a string to the **exec** routine. When programs are executed this way, the executing program does not need to await the finish of the program, nor can it find out if the program ran correctly. If this is required, the routine **execw** is used. **execw** will wait for the program to complete, then place the error return for the program in a variable. This variable will be 0 if the program ran correctly, or non-zero if it didn't. The exact numerical meaning of the error is up to the program executed.

If the system cannot execute programs in parallel, **exec** is equivalent to **execw**, but without the return code.

If the environment is to be set for the executed program, the call **exece** can be used, which takes an environment list. This allows the environment to be retrieved from the current environment or created as new, modified or added to as needed, then passed to the executed program. **execew** does the same thing, but waits for the program to finish, and returns an error code.

If a command line concept does not exist on the target system, an implementation can pass it via another means, such as a file or environmental variable. Alternately, it could simply be ignored.

### H.8 Error Return Code

Each program, when it completes, returns an error code. By convention, if the error code is 0, then no error occurred. If the code is not 0, then an error occurred, and the meaning of the number is defined by the application.

When a program under Pascaline exits, it returns a code that was set to 0 by default when the program started. The procedure **seterr** can be used to set a non-zero code, which will be returned when the program exits. Note that it does ***not*** cause the current program to exit, it simply sets the code that will be returned when it does. It does not matter how the program exits, from the main program block, or a **halt** statement or exception.

### H.9 Creating or Removing Paths

A file path, or directory, is created by **makpth**, and removed by **rempth**. If the directory has files in it, then it cannot be removed until all the files (and directories) under it have been removed. This means by implication that each section of a path must be removed separately, and a tree structured delete would have to repeatedly remove the contents of one element of the path, then remove the path section itself, and so on.

### H.10 Option Character

When parsing commands, the option character for the current operating system is found with **optchr**. **Services** does not define the exact format of command line options. The option character is an aid to portability.

### H.11 Path Character

The path character is used to separate path components, usually directories in a tree structured file system, within a path for the given system. It can be found with the **pthchr** function. **services** does not define the contents, structure or meaning of a path. The path division character is an aid to portability.

## H.12 Location

The functions **latitude** and **longitude** exist to give the location of the host computer in geographic coordinates, and return integers. The measurements are ratioed to **maxint**.

The **longitude** is 0 at the Prime Meridian, a line passing thought Greenwich, UK. The longitudes 0 to **maxint** are east of the line (or in the future timewise), and longitudes 0 to –**maxint** are west of the line. Thus **maxint** and –**maxint** meet on the opposite side of the world from Greenwich. This means for a 32 bit integer that there is 0.0000000838190317 degrees for each step or 9.3306920025 millimeters per step.

The **latitude** is 0 at the equator and **maxint** at the north pole, and –**maxint** at the south pole.

The **longitude** and **latitude** in minutes and seconds can be found with:

```
program location;

var degrees_longitude: integer;
    minutes_longitude: integer;
    second_longitude: integer;
    degrees_latitude: integer;
    minutes_latitude: integer;
    seconds_latitude: integer;
    west: boolean;
    north: boolean;

begin

   degrees_longitude := round(longitude * 0.0000000838190317);
   minutes_longitude := round(xlongitude mod
                              11930465/198841.078518519);
   seconds_longitude := round(longitude mod
                              198841/3314,0179753086400000);
   if degrees< 0 then west := true else west := false;

   degrees_latitude := round(longitude *
                            0.0000000838190317);
   minutes_latitude := round(longitude mod
                              11930465/198841.078518519);
   seconds_latitude := round(longitude mod
                              198841/3314,0179753086400000);
   if degrees_latitude < 0 then north := false else north := true

end.

[need to correct this for ellipsoid]
```

The shape of the world is approximated as an ellipsoid. This means that the circumference of the earth at the equator is a longer distance than the circumference of the earth on the prime meridian (by about 68 kilometers).

The altitude of the host in MSL or Mean Sea Level is given by **altitude**. It is 0 for the mean surface of the ocean (sea level averaged over a long period of time), and **maxint** at 100 kilometers in height (the altitude at which space begins). It is –**maxint** 100 kilometers in depth.

The altitude of 0 (MSL) is typically defined as height above a model of the geoid, or idealized model of the earth. This means it would have to be calculated against the latitude and longitude to find the actual distance from true center of the earth. However, in the majority of cases it can be accepted as a relative measurement.

The location of the host can be entered by the user or determined automatically by instrumentation (GPS). The host could even be mobile, in which case it is possible to determine speed and direction from the coordinates against time.

If there is no location, it is indicated by longitude equal to –**maxint**. This value of **longitude** is redundant to **maxint**. Both **longitude** and **latitude** are considered unavailable by the value of **longitude**.

**Altitude** is available separate from **longitude** and **latitude**. It is not available when **altitude** is –**maxint**.

The country of location is found with **country**, which gives a string corresponding to the English name of the country. At this writing, the following countries exist, in alphabetical order:

| # | Name | # | Name | # | Name | # | Name |
|---|---|---|---|---|---|---|---|
| **1** | Afghanistan | 4 | Dhekelia | 131 | Kyrgyzstan | 196 | Saint Kitts and Nevis |
| **2** | Aland Islands | 248 | Djibouti | 132 | Laos | 197 | Saint Lucia |
| **3** | Albania | 8 | Dominica | 133 | Latvia | 198 | Saint Pierre and Miquelon |
| **4** | Algeria | 12 | Dominican Republic | 134 | Lebanon | 199 | Saint Vincent and the Grenadines |
| **5** | American Samoa | 16 | Ecuador | 135 | Lesotho | 200 | Samoa |
| **6** | Andorra | 20 | Egypt | 136 | Liberia | 201 | San Marino |
| **7** | Angola | 24 | El Salvador | 137 | Libya | 202 | Sao Tome and Principe |
| **8** | Anguilla | 660 | Equatorial Guinea | 138 | Liechtenstein | 203 | Saudi Arabia |
| **9** | Antarctica | 10 | Eritrea | 139 | Lithuania | 204 | Senegal |
| **10** | Antigua and Barbuda | 28 | Estonia | 140 | Luxembourg | 205 | Serbia and Montenegro |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **11** | Argentina | 32 | Ethiopia | 141 | Macau | 206 | Seychelles |
| **12** | Armenia | 51 | Europa Island | 142 | Macedonia | 207 | Sierra Leone |
| **13** | Aruba | 533 | Falkland Islands (Islas Malvinas) | 143 | Madagascar | 208 | Singapore |
| **14** | | | Faroe Islands | 144 | Malawi | 209 | Slovakia |
| **15** | Australia | 36 | Fiji | 145 | Malaysia | 210 | Slovenia |
| **16** | Austria | 40 | Finland | 146 | Maldives | 211 | Solomon Islands |
| **17** | Azerbaijan | 31 | France | 147 | Mali | 212 | Somalia |
| **18** | Bahamas, The | 44 | French Guiana | 148 | Malta | 213 | South Africa |
| **19** | Bahrain | 48 | French Polynesia | 149 | Marshall Islands | 214 | South Georgia and the South Sandwich Islands |
| **20** | Bangladesh | 85 | French Southern and Antarctic Lands | 150 | Martinique | 215 | Spain |
| **21** | Barbados | 86 | Gabon | 151 | Mauritania | 216 | Spratly Islands |
| **22** | Bassas da India | 87 | Gambia, The | 152 | Mauritius | 217 | Sri Lanka |
| **23** | Belarus | 88 | Gaza Strip | 153 | Mayotte | 218 | Sudan |
| **24** | Belgium | 89 | Georgia | 154 | Mexico | 219 | Suriname |
| **25** | Belize | 90 | Germany | 155 | Micronesia, Federated States of | 220 | Svalbard |
| **26** | Benin | 91 | Ghana | 156 | Moldova | 221 | Swaziland |
| **27** | Bermuda | 92 | Gibraltar | 157 | Monaco | 222 | Sweden |
| **28** | Bhutan | 93 | Glorioso Islands | 158 | Mongolia | 223 | Switzerland |
| **29** | Bolivia | 94 | Greece | 159 | Montserrat | 224 | Syria |
| **30** | Bosnia and Herzegovina | 95 | Greenland | 160 | Morocco | 225 | Taiwan |
| **31** | Botswana | 96 | Grenada | 161 | Mozambique | 226 | Tajikistan |
| **32** | Bouvet Island | 97 | Guadeloupe | 162 | Namibia | 227 | Tanzania |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **33** | Brazil | 98 | Guam | 163 | Nauru | 228 | Thailand |
| **34** | British Indian Ocean Territory | 99 | Guatemala | 164 | Navassa Island | 229 | Timor-Leste |
| **35** | British Virgin Islands | 100 | Guernsey | 165 | Nepal | 230 | Togo |
| **36** | Brunei | 101 | Guinea | 166 | Netherlands | 231 | Tokelau |
| **37** | Bulgaria | 102 | Guinea-Bissau | 167 | Netherlands Antilles | 232 | Tonga |
| **38** | Burkina Faso | 103 | Guyana | 168 | New Caledonia | 233 | Trinidad and Tobago |
| **39** | Burma | 104 | Haiti | 169 | New Zealand | 234 | Tromelin Island |
| **40** | Burundi | 105 | Heard Island and McDonald Islands | 170 | Nicaragua | 235 | Tunisia |
| **41** | Cambodia | 106 | Holy See (Vatican City) | 171 | Niger | 236 | Turkey |
| **42** | Cameroon | 107 | Honduras | 172 | Nigeria | 237 | Turkmenistan |
| **43** | Canada | 108 | Hong Kong | 173 | Niue | 238 | Turks and Caicos Islands |
| **44** | Cape Verde | 109 | Hungary | 174 | Norfolk Island | 239 | Tuvalu |
| **45** | Cayman Islands | 110 | Iceland | 175 | Northern Mariana Islands | 240 | Uganda |
| **46** | Central African Republic | 111 | India | 176 | Norway | 241 | Ukraine |
| **47** | Chad | 112 | Indonesia | 177 | Oman | 242 | United Arab Emirates |
| **48** | Chile | 113 | Iran | 178 | Pakistan | 243 | United Kingdom |
| **49** | China | 114 | Iraq | 179 | Palau | 244 | United States |
| **50** | Christmas Island | 115 | Ireland | 180 | Panama | 245 | Uruguay |
| **51** | Clipperton Island | 116 | Isle of Man | 181 | Papua New Guinea | 246 | Uzbekistan |
| **52** | Cocos (Keeling) Islands | 117 | Israel | 182 | Paracel Islands | 247 | Vanuatu |
| **53** | Colombia | 118 | Italy | 183 | Paraguay | 248 | Venezuela |
| **54** | Comoros | 119 | Jamaica | 184 | Peru | 249 | Vietnam |
| **55** | Congo, Democratic | 120 | Jan Mayen | 185 | Philippines | 250 | Virgin Islands |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | Republic of the | | | | | | |
| **56** | Congo, Republic of the | 121 | Japan | 186 | Pitcairn Islands | 251 | Wake Island |
| **57** | Cook Islands | 122 | Jersey | 187 | Poland | 252 | Wallis and Futuna |
| **58** | Coral Sea Islands | 123 | Jordan | 188 | Portugal | 253 | West Bank |
| **59** | Costa Rica | 124 | Juan de Nova Island | 189 | Puerto Rico | 254 | Western Sahara |
| **60** | Cote d'Ivoire | 125 | Kazakhstan | 190 | Qatar | 255 | Yemen |
| **61** | Croatia | 126 | Kenya | 191 | Reunion | 256 | Zambia |
| **62** | Cuba | 127 | Kiribati | 192 | Romania | 257 | Zimbabwe |
| **63** | Cyprus | 128 | Korea, North | 193 | Russia | | |
| **64** | Czech Republic | 129 | Korea, South | 194 | Rwanda | | |
| **65** | Denmark | 130 | Kuwait | 195 | Saint Helena | | |

If the country is not set, an empty string (not a nil string) is returned.

The ordinal number of the country is given by **countryord**, which returns the number of the language from the table above. These ordinal numbers are given by the standard ISO 3166-1.

The current time zone, is given by **timezone**. It gives hours in the range of -12 to +14, which indicate the offset in hours from GMT or Greenwich Mean Time. The function **daylightsav** is true if daylight savings is in effect in the current host location.

If 24 hour time is used in the current host location, the function **time24hour** will return **true**, otherwise **false**. The accepted format for 24 hour time is with hours from 0 to 23.

Both the current time zone and daylight savings time are factored into the calculation of **local**, and that is the preferred method to find local time.

## H.13 Internationalization

The current language in use on the host count be found with **language**, which gives a string corresponding to the English name of the language. The languages used are according to the ISO 639-1 standard:

| # | Name | # | Name | # | Name | # | Name |
|---|---|---|---|---|---|---|---|
| 1 | Afan | 36 | French | 71 | Lithuanian | 106 | Siswati |
| 2 | Abkhazian | 37 | Frisian | 72 | Macedonian | 107 | Slovak |
| 3 | Afar | 38 | Galician | 73 | Malagasy | 108 | Slovenian |
| 4 | Afrikaans | 39 | Georgian | 74 | Malay | 109 | Somali |
| 5 | Albanian | 40 | German | 75 | Malayalam | 110 | Spanish |
| 6 | Amharic | 41 | Greek | 76 | Maltese | 111 | Sudanese |
| 7 | Arabic | 42 | Greenlandic | 77 | Maori | 112 | Swahili |
| 8 | Armenian | 43 | Guarani | 78 | Marathi | 113 | Swedish |
| 9 | Assamese | 44 | Gujarati | 79 | Moldavian | 114 | Tagalog |
| 0 | Aymara | 45 | Hausa | 80 | Mongolian | 115 | Tajik |
| 11 | Azerbaijani | 46 | Hebrew | 81 | Nauru | 116 | Tamil |
| 12 | Bashkir | 47 | Hindi | 82 | Nepali | 117 | Tatar |
| 13 | Basque | 48 | Hungarian | 83 | Norwegian | 118 | Tegulu |
| 14 | Bengali | 49 | Icelandic | 84 | Occitan | 119 | Thai |
| 15 | Bhutani | 50 | Indonesian | 85 | Oriya | 120 | Tibetan |
| 16 | Bihari | 51 | Interlingua | 86 | Pashto | 121 | Tigrinya |
| 17 | Bislama | 52 | Interlingue | 87 | Persian | 122 | Tonga |
| 18 | Breton | 53 | Inupiak | 88 | Polish | 123 | Tsonga |
| 19 | Bulgarian | 54 | Inuktitut | 89 | Portuguese | 124 | Turkish |
| 20 | Burmese | 55 | Irish | 90 | Punjabi | 125 | Turkmen |
| 21 | Byelorussian | 56 | Italian | 91 | Quechua | 126 | Twi |
| 22 | Cambodian | 57 | Japanese | 92 | Rhaeto-Romance | 127 | Uigur |
| 23 | Catalan | 58 | Javanese | 93 | Romanian | 128 | Ukrainian |
| 24 | Chinese | 59 | Kannada | 94 | Russian | 129 | Urdu |
| 25 | Corsican | 60 | Kashmiri | 95 | Samoan | 130 | Uzbek |
| 26 | Croatian | 61 | Kazakh | 96 | Sangro | 131 | Vietnamese |
| 27 | Czech | 62 | Kinyarwanda | 97 | Sanskrit | 132 | Volapuk |
| 28 | Danish | 63 | Kirghiz | 98 | ScotsGaelic | 133 | Welch |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **29** | Dutch | 64 | Kirundi | 99 | Serbian | 134 | Wolof |
| **30** | English | 65 | Korean | 100 | Serbo-Croatian | 135 | Xhosa |
| **31** | Esperanto | 66 | Kurdish | 101 | Sesotho | 136 | Yiddish |
| **32** | Estonian | 67 | Laothian | 102 | Setswana | 137 | Yoruba |
| **33** | Faeroese | 68 | Latin | 103 | Shona | 138 | Zhuang |
| **34** | Fiji | 69 | Latvian | 104 | Sindhi | 139 | Zulu |
| **35** | Finnish | 70 | Lingala | 105 | Singhalese | | |

The ordinal number of the language is given by **languageord**, which returns the number of the language from the table above. Note that even if the languages are added to or subtracted to in future implementations, the ordinal numbers will not be changed, simply extended at the end, with any removed entries set to an empty string.

The decimal point character in use can be found with **decimal**. The current number separator in use is defined with **numberseparator**.

The time and date formats can be derived the country of the host. The main difference between formats is the order of the elements in time and date. The functions **timeorder** and **dateorder** give the ordering:

| dateorder Code | Date format |
|---|---|
| **1** | year-month-day (ISO 8601 standard format) |
| **2** | year-day-month |
| **3** | month-day-year |
| **4** | month-year-day |
| **5** | day-month-year |
| **6** | day-year-month |

| timeorder Code | Time format |
|---|---|
| **1** | Hour:minute:second (ISO 8601 standard format. |
| **2** | Hour:second:minute |
| **3** | Minute:hour:second |
| **4** | Minute:second:hour |
| **5** | Second:hour:minute |
| **6** | Second:minute:hour |

The separator character for fields in the date is given by **dateseparator**, which is '-' in ISO 8601 date formats. The separator character for fields in time is given by **timeseparator**, which is ':' in ISO 8601 time formats.

The number of digits in each section of the time and date formats is:

| Section | Digits |
|---|---|
| **Year** | 4 |
| **Month** | 2 |
| **Day** | 2 |
| **Hour** | 2 |
| **Minute** | 2 |
| **Second** | 2 |

If the time and date format is not set, it defaults to the ISO 8601 standard for time and date formatting. This is the correct format for output that can be read across international boundaries.

Note that the procedures **times**, **dates**, **writetime** and **writedate** automatically use the internationalization settings of the host to arrive at the host computers natural time and date formatting.

The symbol for the currency used in the country of host is given by **currencychr**.

## H.14 Exceptions

The following exceptions are generated in **services**:

| Identifier | Meaning |
|---|---|
| **NameTooLong** | The filename passed was too long for a **services** internal buffer. |
| **FileSizeTooLarge** | The file(s) being processed were too large. |
| **StringNil** | The string passed was nil. |
| **StringTooSmallForTime** | The string result buffer was too small to hold the time. |
| **StringTooSmallForDate** | The string result buffer was too small to hold the date. |
| **EnvironmentStringTooLarge** | A system environment string was too large for a **services** buffer. |
| **CommandStringEmpty** | The command string passed was empty. |
| **ProgramNotFound** | External program not found. |
| **CurrentPathTooLong** | Current path is too long for a **services** buffer. |
| **FilenameStringEmpty** | The filename passed was empty. |
| **CannotDetermineProgramPath** | The program path could not be determined. |

**Services** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **services** will go back to **services**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **services** need only capture the exceptions occurring in the procedure or function that is called.

## H.15 Functions and procedures in services

For all of the following calls, where an output file is used, it can be left off. The result is to default to the standard **output** file.

procedure list(fn: [p]string; var l: filptr);

Form a file list from the filename **fn**, and return in the file entry list **l**. The filename **fn** may contain wildcards, and may be fully pathed, or refer to the current directory. If no files are found, then the list pointer is returned **nil**.

Exceptions: **FileSizeTooLarge**, **StringNil**

procedure times(var s: string; t: linteger);
function times(t: linteger): pstring;

Place time from S2000 time **t** in string **s**.

Exceptions: **StringTooSmallForTime**

procedure dates(var s: string; t: linteger);
function dates(t: linteger): pstring;

Place date from S2000 time **t** in string **s**.

Exceptions: **StringTooSmallForDate**

procedure writetime([var f: text;] t: linteger);

Write time from S2000 time **t** to text file **f**, or by default, the standard output file.

Exceptions: None

procedure writedate([var f: text;] t: linteger);

Write date from S2000 time **t** to text file **f**, or by default, the standard output file.

Exceptions: None

function time: linteger;

Returns current S2000 time in GMT

Exceptions: None

function local(t: linteger): linteger;

Converts the given GMT S2000 time **t** to local time, using the time zone offset and daylight savings status in the host computer, and returns the result.

Exceptions: None

function clock: lcardinal;

Returns 100 microsecond, free running time.

Exceptions: None

function elapsed(r: lcardinal): lcardinal;

Given a stored **clock** time **r**, will check the current **clock** time and find the total number of 100 microsecond ticks since the given time, then return that. Accounts for timer wraparound.

Exceptions: None

function validfile(s: [p]string): boolean;

Parses and checks the file specification **s** for a valid filename on the current system. Returns true if valid.

Exceptions: **StringNil**

function validpath(s: [p]string): boolean;

Parses and checks the file specification **s** for a valid path on the current system. Returns true if valid, otherwise false.

Exceptions: **StringNil**

function wild(s: [p]string): boolean;

Checks if the file specification **s** contains wildcards. Returns true if so, otherwise false.

Exceptions: **StringNil**

procedure getenv(ls[p]: string; var ds: string);
function getenv(ls[p]: string): pstring;

Finds and returns an environment string. **ls** contains the name of the string to look up, **ds** or the return value contains the resulting string as found. If there is no environment string by that name, the return is either all blanks, or nil.

Exceptions: **EnvironmentStringTooLarge**

procedure setenv(sn: [p]string; sd: [p]string);

Finds and sets an environment string. **sn** contains the string name to set, and **sd** contains the contents to set it to. If there is no string by that name, it is created, otherwise the old string is replaced.

Exceptions: **StringNil**

procedure allenv(out el: envptr);

Returns a complete list of the strings in the environment to **el**.

Exceptions: None

procedure remenv(sn: [p]string);

Remove a string **sn** from the environment. The string is found by name, and removed from the environment. No error results if the string does not exist.

Exceptions: **StringNil**

procedure exec(cmd: [p]string);

Execute external program, with parameters, from the string **cmd**. Does not wait for the program to finish, and cannot detect if it finished with an error.

Exceptions: **CommandStringEmpty**, **ProgramNotFound**, **StringNil**

procedure exece(cmd: [p]string; el: envptr);

Execute external program, with parameters from the string **cmd** and full environment. The environment is passed as a list in **el**. Does not wait for the program to finish, and cannot detect if it finished with an error.

Exceptions: **CommandStringEmpty**, **ProgramNotFound**, **StringNil**

procedure execw(cmd: [p]string; var e: integer);

Execute external program, with parameters from the string **cmd**. Waits for the program to finish, and returns its error code in **e**. The error code is 0 for no error, otherwise the error is a code specified by the program executed.

Exceptions: **CommandStringEmpty**, **ProgramNotFound**, **StringNil**

procedure execew(cmd: [p]string; el: envptr; var e: integer);

Execute external program, with parameters from the string **cmd** and full environment. The environment is passed as a list in **el**. Waits for the program to finish, and returns its error code in **e**. The error code is 0 for no error, otherwise the error is a code specified by the program executed.

Exceptions: **CommandStringEmpty**, **ProgramNotFound**, **StringNil**

procedure getcur(var fn: string);
function getcur: pstring;

Get current path to **fn** or the result. Returns the current directory path.

Exceptions: **CurrentPathTooLong**

procedure setcur(fn: [p]string);

Set current path from string **fn**. Sets the default path for all file specifications.

Exceptions: **StringNil**

procedure getpgm(var s: string);
function getpgm: pstring;

Get the program path to **s** or returns it. Returns the program path, which is the path the program running was loaded from.

Exceptions: **CannotDetermineProgramPath**

procedure getusr(var fn: string);
function getusr: pstring;

Get user path to **fn** or returns it. Return the user path, which is a path specific to each user.

Exceptions: None

procedure brknam(fn: [p]string; var p, n, e: [p]string);

Break down file specification. Breaks the file specification **fn** down into path **p**, name **n**, and extension **e**. Note that any one of the resulting components could be blank, if it does not exist in the name.

Exceptions: **FilenameStringEmpty**, **StringNil**

procedure maknam(var fn: string; view p, n, e: string);
function maknam(p: [p]string; n: [p]string; e: [p]string): pstring;

Create file specification from components. Creates file specification **fn** from path **p**, name **n**, and extension **e**. Components may be blank, but the path and the name cannot both be blank.

Exceptions: **NameTooLong**, **StringNil**

procedure fulnam(var fn: string);
function fulnam(fn: string): pstring;

Create full file specification from a partial file specification **fn**. Given a file specification with an incomplete path (either by using the default path, or mnemonic shortcuts for things like parent directory), creates a fully pathed name of standard form. This can "normalize" file specifications, for comparisons, and to store the complete path for the file. The fully pathed result is returned in **fn** or as the function result.

Exceptions: None

procedure setatr(fn: [p]string; a: attrset);

Set attributes. Given a file by name **fn**, the attributes in the set **a** are set true for the file.

Exceptions: **StringNil**

procedure resatr(fn: [p]string; a: attrset);

Reset attributes. Given a file by name **fn**, the attributes in the set **a** are set false for the file.

Exceptions: **StringNil**

procedure bakupd(fn: [p]string);

Set backup time current. Given a file by name **fn**, sets the backup time for the file as current. Backup programs should also reset the archive bit to show that backup has occurred.

Exceptions: **StringNil**

procedure setuper(fn: [p]string; p: permset);

Set user permissions. Given a file by name **fn**, the permissions in the set **p** are set true for the file.

Exceptions: **StringNil**

procedure resuper(fn: [p]string; p: permset);

Reset user permissions. Given a file by name **fn**, the permissions in the set **p** are set false for the file.

Exceptions: **StringNil**

procedure setgper(fn: [p]string; p: permset);

Set group permissions. Given a file by name **fn**, the permissions in the set **p** are set true for the file.

Exceptions: **StringNil**

procedure resgper(fn: [p]string; p: permset);

Reset group permissions. Given a file by name **fn**, the permissions in the set **p** are set false for the file.

Exceptions: **StringNil**

procedure setoper(fn: [p]string; p: permset);

Set other permissions. Given a file by name **fn**, the permissions in the set **p** are set true for the file.

Exceptions: **StringNil**

procedure resoper(fn: [p]string; p: permset);

Reset group permissions. Given a file by name **fn**, the permissions in the set **p** are set false for the file.

Exceptions: **StringNil**

procedure seterr(e: integer);

Set program return error **e**. The error code returned by the current program is set. This has no effect until the program exits.

Exceptions: None

procedure makpth(fn: [p]string);

Make path using **fn**. Creates a new path or directory. If the path already exists, it's an error.

Exceptions: **StringNil**

procedure rempth(fn: [p]string);

Remove path **fn**. Removes a path, or directory. The directory must exist, and must be empty of any files or other directories, or an error results.

Exceptions: **StringNil**

procedure filchr(var fc: chrset);

Returns the set of valid filename characters in **fc**.

Exceptions: None

function optchr: char;

Find the option character. Returns the character that is used to introduce options in command liens on the current system.

Exceptions: None

function pthchr: char;

Returns the character used to separate components in a path in the current system.

Exceptions: None

# I Annex: String Library

The string library implements various useful string functions. Strings are built out of arrays of characters in Pascal, and extended in Pascaline. **strings** unifies two schemes. The first is the space padded right scheme familiar from ISO 7185 Pascal. The second is dynamic strings.

Dynamic strings are almost the ideal string type. They are unlimited in length, they can be returned from a function. And because their storage is recycled, they are space efficient. They can also be fairly speed efficient by managing when and if they are recycled.

The system declarations for padded or fixed, and dynamic strings appear as:

```
type string  = packed array of char;
     pstring = ^string;
```

The string functions are often overloaded so that they take both types of strings.

## I.1 Conventions

In some cases, it is ambiguous whether a padded or dynamic string argument is meant. For example, the string compare facility does not know which type its operands are, and it makes a difference to the result, since length plays a part in string comparison. For these situations, a "p" (for padded) is appended to the name of the procedure or function.

Many functions and procedures must know if case matters. For example, string compare can be with case, or caseless. For these procedures and functions, there is appended a "c" to the ones that case does matter.

A few functions and procedures perform different actions on strings vs. characters. For the string versions, an "s" is appended to the name.

## I.2 Words

Some of the functions and procedures treat the strings as a series of words. Words are a series of non-space characters surrounded by one or more spaces. For example:

```
‘    hi       there                George          ‘
```

Has three words, 'hi', 'there', and 'george'. Such words can be counted, indexed and extracted.

## I.3 Format Strings

Some routines accept a "format string" to output numbers with. The format is an “image” of the output string the number is converted into. The string will contain a series of format characters. The entire format string is copied to the result string, but the special format characters are replaced with parts of the number to be converted.

The number “image” contained within the format string consists of any number of contiguous characters from the set [‘9’, ‘0’, ’-‘, ‘+’, ‘$’, ‘&’ ‘%’,’,‘]. Any number of other characters can appear before the image, and any number of other characters can follow it.

The first thing the format routines do is match the format to the number for decimal point position. For example:

```
+999,999.999       Format characters
      50.12        Actual number (without leading or trailing
                   zeros)
```

Then, the meaning of each format character depends on if that character appears to the left or to the right of the decimal point, and if there are non-zero digits more significant (if to the left of the decimal point) or non-zero digits less significant (if to the right of the decimal point).

If a decimal point is seen, and one has already appeared in the current number image, it is an error.

Each format character operates on the digit that matches it’s position within the actual number after decimal point alignment.

The format characters are:

9

Represents a digit. To the left of the decimal point, this is replaced with its matching digit from the number. If there is no digit in the number at that position, and no more significant non-zero digit, a space replaces the format character. To the right of the decimal point, if there is no digit at that position, and no less significant non-zero digit, a space replaces the format character.

0

As "9" above, but "0" replaces the digit if no significant digit is found, instead of space when to the left of the decimal point. To the right of the decimal point, if the digit position is non-zero, then it replaces the format character, otherwise remains 0.

-

Represents the sign. To the left of the decimal point, if the number is negative, it is left alone. If the number is positive, it is replaced by a space. If the sign was already output, then it is replaced by space. If this appears to the right of the decimal point, it terminates the image.

+

As "-", but "+" appears instead of space on a positive number.

$, &, %

These characters are used to indicate if the number is hex (or USA dollars), octal or binary. To the left of the decimal point, if a significant digit can be matched to this position, the digit is

output, otherwise, either the format character, or a space is placed. A space is placed if any of the format characters ‘$’, ‘&’, or ‘%’ have already appeared. It is an error if this format character is used to the right of the decimal point.

,

To the left of the decimal point, if the comma appears to the right of any non-zero significant digits, it is output as is. Otherwise, it is either replaced by space. If a "$", "&" or "%" character appears to the left of it, the format character will appear. It is an error if this format character is used to the right of the decimal point.

.

May only appear on real numbers. This format character is always printed, but specifies where the mantissa of a number appears, and where its fraction appears. The appearance of the decimal point will enable or disable the fraction. If present, fractional digits are output, otherwise, the fraction is discarded.

## I.4 Recycling

By default, **strings** leaves it up to the programmer to determine when dynamic strings should be recycled. To keep from losing space ("memory leaks"), the program must be careful to keep track of all dynamic strings created, and return them to free storage via **dispose**.

**strings** can use a nested blocking system to automatically dispose of strings for you. The **openstring** call begins a new string block, and **closestring** ends it. When a new block is opened, any dynamic strings allocated are recorded in the block. Then, each of them are disposed of when the block ends. Any number of block levels can exist. With no blocks in effect, the automatic recycling system is off, which is the state **strings** starts in. When all blocks are closed, **strings** reverts to this state. Because the dynamic strings are returned as a group, this system can be more efficient than returning the strings one at a time.

Use **exportstring** to completely remove a given dynamic string from the automated recycling system. The **upstring** call takes a string, and moves it to the surrounding string block.

The **strings** block system has nothing to do with blocks in Pascal. **strings** blocks can cross blocks, functions and even whole modules of Pascal. The two are entirely unrelated.

## I.5 String container classes

The functionality of the **strings** library is available in a **stringc** class:

```
module strings;

class stringc(l: integer);

type stringcr: reference to stringc;

var val(l): string;

procedure lcases; begin end;
procedure ucases; begin end;
procedure clears; begin end;
function len: integer;
overload procedure len(newlen: integer);
procedure copy(s: pstring);
overload procedure copy(var s: string);
overload procedure copy(var s: stringc);
overload procedure copy(s: stringcr);
operator := (s: string);
operator := (s: pstring);
operator := (s: stringc);
operator := (s: stringcr);
operator := (sd: string; ss: stringc);
operator := (sd: pstring; ss: stringc);
operator := (sd: string; ss: stringcr);
operator := (sd: pstring; ss: stringcr);
procedure cat(s: string);
procedure cat(s: pstring
procedure cat(var s: stringc);
procedure cat(s: stringcr);
operator + (s:string): stringcr;
operator + (s: pstring): stringcr;
operator + (var s: stringc);
operator + (s: stringcr);



function comp[c]p(s: string): boolean;
function comp[c]p(s: pstring): boolean;
function comp[c]p(s:[p]string): boolean;
function comp[c]p(s:[p]string): boolean;
function comp[c]p(s:[p]string): boolean;




function gtr[c]p(s: [p]string): boolean;
operator > (s: [p]string): boolean;
operator < (s: [p]string): boolean;
operator = (s: [p]string): boolean;
```

```
operator >= (s: [p]pstring): boolean;
operator <= (s: [p]string): boolean;
function index[c]p(s: [p]string): integer;
procedure extract(l, r: integer);
procedure insert(s: string; p: integer);
procedure rep(s:string; r: integer);
operator * (r: integer): stringcr;
procedure trim;
function words: integer;
procedure extwords(l, r: integer);
procedure reads([f: text][var ovf: boolean]);
procedure ints(i: integer [; fl: integer] | [fmt: string]);
procedure reals(r: real [; f: integer] | [; fl: integer; fr:
integer] | [; fmt: string]);
procedure reales(r: real [; fl: integer]);
procedure hexs(w: lcardinal [; fl: integer]);
procedure octs(w: lcardinal [; fl: integer]);
procedure bins(w: lcardinal [; fl: integer]);
function intv][(var ovf: boolean)]: integer;
function hexv[(var ovf: boolean)]: lcardinal;
function octv[(var ovf: boolean)]: lcardinal;
function binv[(var ovf: boolean])]: lcardinal;
function realv: real;
procedure subst[c][all](s: [p]string; r: [p]string);

.

begin ! strings
end.
```

### I.6 Exceptions

The following exceptions are generated in **strings**:

| Identifier | Meaning |
|---|---|
| **NoStringBlock** | No string block is active. |
| **OuterBlockFull** | No room in outermost string block. |
| **CurrentBlockFull** | Current string block is full. |
| **StringNil** | String passed is nil. |
| **StringDestinationOverflow** | String was to large for destination. |
| **IndexOutOfRange** | String index out of range. |
| **NegativeRepeatCount** | Repeat count was negative. |
| **WordIndexOutOfRange** | Word array index was out of range. |
| **StringReadOverflow** | String was too large to read. |
| **FormatTooLarge** | Format is too large for destination |
| **InvalidFieldSpecification** | Field specified is invalid |
| **NegativeValueNondecimal** | Radix was negative |
| **NumberOverflowsFormat** | Number overflows space provided in format string. |
| **NegativeNotPlaced** | Negative sign not placed in format. |
| **InvalidRealNumber** | Invalid real number. |
| **InvalidFractionSpecification** | Invalid fraction specification. |
| **InvalidRadix** | Invalid radix |
| **InvalidIntegerFormat** | Invalid integer format. |
| **NumberTooLarge** | Number too large. |
| **IntegerTooLarge** | Integer too large. |
| **InvalidRealFormat** | Invalid real format. |

**strings** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **strings** will go back to **strings**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **strings** need only capture the exceptions occurring in the procedure or function that is called.

## I.7 Procedures and functions in strings

For each call that uses a file for input or output, the file can be left off. The default is the standard **input** or **output** file, accordingly.

```
function lcase(c: char): char;
function lcase(s: pstring): pstring;
function lcases(s: string): pstring;
procedure lcases(s: string);
```

---

Finds the lower case version of a character **c** or string **s**. This is either returned, or converted in place.

Exceptions: **StringNil**

function ucase(c: char): char;
function ucase(s: pstring): pstring;
function ucases(s: string): pstring;
procedure ucases(s: string);

Finds the upper case version of a character **c** or string **s**. This is either returned, or converted in place.

Exceptions: **StringNil**

procedure clears(var s: string);

Clears string **s** to all spaces. To set an entire string to another character value besides space, use **rep** below.

Exceptions: None

function len(s: [p]string): integer;

Finds the padded length of a string **s**. This is equivalent to the index of the last non-space character, or zero if there is none. Note that dynamic string lengths are found via the system function **max**.

Exceptions: **StringNil**

procedure copy(var d: string; s: [p]string);
function copy(s: [p]string): pstring;
procedure copy(var d: pstring; s: string);
operator := (var d: string; s: pstring);
operator := (var d: pstring; s: string);

Create a copy of the source **s** string in the destination **d** or returns it as a result. If the destination is a fixed string, then it will be padded on the right with spaces to fill it out. If the destination or return type is dynamic, then a new string will be created of the same length as the source, and the source copied to that. In the case of the last procedure, the source is taken to be a padded string, and the length is without padding. This procedure is used to copy from a padded string to a dynamic string.

Note that the assignment operator := is defined in the case of string to string assignment as to copy the contents of the strings if the strings are of equal length, and it is an error if they are not of the same length. In the case of pstring to pstring, ISO 7185 defines only the pointer to be copied. Thus, both the source and destination will simply point to the same string.

Exceptions: **StringNil**, **StringDestinationOverflow, CurrentBlockFull**

procedure cat(var d: string; s: string);
function cat(sa: [p]string; sb:[p]string): pstring;
operator + (sa: [p]string; sb:[p]string): pstring;

Concatenate two strings **d** and **s**, or **sa** and **sb**. For the first procedure, the padded destination and source are concatenated into the padded destination. The function concatenates unpadded strings to a dynamic result.

When concatenating two padded strings, it is not possible using cat to concatenate with a space or spaces between strings. For this case, use a padded string insert instead, with the source string inserted after the desired number of spaces.

Exceptions: **StringNil**, **StringDestinationOverflow, CurrentBlockFull**

function comp[c](sa: [p]string; sb: [p]string): boolean;
function comp[c]p(sa, sb:string): boolean;

Compare strings **sa** and **sb**. Returns true if the strings are equal. Compares case or caseless, and padded or fixed length. Strings are equal only if they have the same length and content.

Note that the overload equals and not equals version of copy cannot be used with a ISO 7185 Pascal string, since those operators are already defined.

Exceptions: **StringNil**

function gtr[c](sa: [p]string; sb: [p]string): boolean;
function gtr[c]p(sa, sb:string): boolean;

Compare string **sb** is greater than string **sa**. Returns true if the second string is greater than the first. Compares case or caseless, and padded or fixed length. Strings are compared from left to right, until the first difference is found. Then, the second string is greater than the first if the ord of its character is greater than the first. In case one string is longer than the other, but otherwise equal, the shorter string is less than the longer string.

This function can be used to find less than, greater than or equal, and less than or equal by arranging the operands:

```
gtr(a, b)      a < b
gtr(b, a)      a > b
not gtr(a, b)  a >= b
not gtr(b, a)  a <= b
```

Exceptions: **StringNil**

operator > (sa: string; sb: pstring): boolean;
operator > (sa: pstring; sb: string): boolean;
operator > (sa: ptring; sb: pstring): boolean;
operator < (sa: string; sb: pstring): boolean;
operator < (sa: pstring; sb: string): boolean;
operator < (sa: ptring; sb: pstring): boolean;
operator = (sa: string; sb: pstring): boolean;
operator = (sa: pstring; sb: string): boolean;
operator = (sa: ptring; sb: pstring): boolean;
operator >= (sa: string; sb: pstring): boolean;
operator >= (sa: pstring; sb: string): boolean;
operator > =(sa: ptring; sb: pstring): boolean;
operator <= (sa: string; sb: pstring): boolean;
operator <= (sa: pstring; sb: string): boolean;
operator <= (sa: ptring; sb: pstring): boolean;

Finds string greater than, less than, equals, not equals, greater or equals, and less than or equals for any combination of pstring with string, or between two pstrings. The comparision is always considering the case of the string characters. The length of the strings is according the absolute length of the strings without considering padding.

For equals, two strings are equal if they have the same characters at the same positions, and if they are the same length.

For other relations, the strings are examined character by character until a difference is found, or the end of both strings is reached. If a difference is found, the ordering of the character found to be different determines the result. If the strings are found to be different lengths, the longer string is greater than the shorter one.

Note that the operators for ISO 7185 Pascal strings are defined within the ISO 7185 standard, and only allow strings of the same length to be compared. Only in the above functions is the length rule relaxed to allow strings of different lengths to be compared.

If a comparison is needed that does not consider case, or uses the padded length definition of strings is needed, use the **comp** and **gtr** functions listed above.

Exceptions: **StringNil**

function index[c](sa: [p]string; sb: [p]string): integer;
function index[c]p(sa, sb:string): integer;

Finds the incidence of the source string **sb** in the string **sa**. If the source string is found within the destination, the index of its first character is returned, otherwise 0. The comparison may be case or caseless. The padded version is for when the source string is padded.

Exceptions: **StringNil**

procedure extract(var d: string; s: string; l, r: integer);
function extract(s: [p]string; l, r: integer): pstring;

Extract a substring. The source string **s** from the left index to the right index is extracted, and either placed in the destination **d**, or returned. It is an error if either index is out of range, but indexes l > r simply result in a null string. If the result is too large for the destination, an error results. The procedure version is for padded strings.

Common equivalents using extract are:

```
right(s,l)  extract(s,max(s)-l+1,max(s))  Get right string
left(s,l)   extract(s,1,l)                Get  left string
mid(s,l,r)  extract(s,l,l+r-1)            Get mid string
```

Exceptions: **StringNil**, **StringDestinationOverflow**, **IndexOutOfRange, CurrentBlockFull**

procedure insert(var d: string; s: string; p: integer);
function insert(sa:[p]string; sb: [p]string; p: integer): pstring;

Inserts a substring from the source **s** into the destination **d**. The procedure version is for padded strings. If the source string is too long for the destination, an error results.

Exceptions: **StringNil**, **StringDestinationOverflow**, **IndexOutOfRange, CurrentBlockFull**

function rep(s: [p]string; r: integer): pstring;
procedure rep(var d: string; s:string; r: integer);
operator * (var d: string; s:string; r: integer);

Repeats the source string **s r** times into the destination **d** or the return string. The procedure version is for padded strings. If the resulting string is too long for the destination, an error results.

Exceptions: **StringNil**, **StringDestinationOverflow, NegativeRepeatCount, CurrentBlockFull**

function trim(s: [p]string): pstring;
procedure trim(var d: string; s: string);

Trim leading and trailing spaces from string **s**. and returns the result in string **d** or the result The procedure version is for padded strings. For padded strings, only the leading spaces are affected.

Exceptions: **StringNil**, **StringDestinationOverflow, IndexOutOfRange, CurrentBlockFull**

function words(s: [p]string): integer;

Returns a count of the number of space delimited words in the string **s**.

Exceptions: **StringNil, WordIndexOutOfRange**

function extwords(s: [p]string; l, r: integer): pstring;
procedure extwords(var d: string; s: string; l, r: integer);

Extracts the space delimited words from the string **s**, between the left to the right indicies inclusive. Returns the result in **d** or as the result.

Exceptions: **StringNil**, **StringDestinationOverflow, WordIndexOutOfRange, CurrentBlockFull**

procedure reads([f: text;] var s: [p]string [; var ovf: boolean]);

Reads a line to string **s** from text file **f**. The versions of this procedure that have an **ovf** flag return it true if the input line overflowed the string. In this case, the string is returned truncated. The non-**ovf** versions of the procedure simply throw an exception.

If the input file is not specified, then the standard **input** file is used.

Note that the behavior of this function without the **ovf** flag, and catching the **StringDestinationOverflow** exception is similar, but the **ovf** flag version allows the partial left hand side of the input to be recovered.

Exceptions: **StringDestinationOverflow, CurrentBlockFull**

procedure ints(var s: string; i: integer [; fl: integer] | [fmt: string]);
function ints(i: integer [; fl: integer] | [fmt: string]): pstring;

Convert integer **i** to string **s**. The versions with a field **fl** format the number using the field using ISO 7185 Pascal rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" in the main text.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

procedure reals(var s: string; r: real [; f: integer] | [; fl: integer; fr: integer] | [; fmt: string]);
function reals(r: real[; f: integer] | [; fl: integer; fr: integer] | [; fmt: string]): pstring;

Convert real **r** to string **s** with fraction **f**. The versions with a field **fl** format the number using the field using ISO 7185 Pascal rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" in the main text.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, InvalidRealNumber, InvalidFractionSpecification, NegativeNotPlaced, CurrentBlockFull**

procedure reales(var s: string; r: real [; fl: integer]);
function reales(r: real [; fl: integer]): pstring;

Convert real **r** to string **s** using "economy" format. The real is printed in the minimum number of characters possible. If the decimal position can be placed into the number, then it is, and the exponent is removed. All insignificant leading and trailing zeros are removed, and if there are no digits to the right or left of the decimal point, then that is removed.

If the decimal point cannot be placed into the number with less total characters than a full ISO 7185 floating point number format, then the standard floating point format is used.

The versions of this call with a field **fl** fit the result into the field by padding it out with blanks, either on the left for positive fields, or on the right for negative fields. If the number does not fit into the specified field, then all of the required parts of the number are printed. Note that if the ISO 7185 floating point number is chosen as the shortest length number, then the rules for fields are identical to that of normal real output formats.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, CurrentBlockFull**

```
procedure hexs(var s: string; w: lcardinal [; fl: integer]);
procedure hexs(var s: string; w: lcardinal; fmt: string);
function hexs(w: lcardinal [; fl: integer]): pstring;
function hexs(w: lcardinal; fmt: string): pstring;
```

Convert **cardinal w** to string **s**, using the hexadecimal radix system. The versions with a field **fl** format the number using the field using ISO 7185 Pascal rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" in the main text.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NegativeValueNondecimal, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

```
procedure octs(var s: string; w: lcardinal [; fl: integer]);
procedure octs(var s: string; w: lcardinal; fmt: string);
function octs(w: lcardinal [; fl: integer]): pstring;
function octs(w: lcardinal; fmt: string): pstring;
```

Convert **cardinal w** to string **s**, using the octal radix system. The versions with a field **fl** format the number using the field using ISO 7185 Pascal rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" above.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NegativeValueNondecimal, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

procedure bins(var s: string; w: lcardinal [; fl: integer]);
procedure bins(var s: string; w: lcardinal; fmt:string);
function bins(w: lcardinal [; fl: integer]): pstring;
function bins(w: lcardinal; fmt: string): pstring;

Convert **cardinal w** to string **s**, using the binary radix system. The versions with a field **fl** format the number using the field using ISO 7185 Pascal rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" above.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NegativeValueNondecimal, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

procedure writed([var f: text;] i: integer; fmt: string);

Convert **integer i** to character format and output to file **f**, using the decimal radix system. Decimal numbers are well covered by the normal ISO 7185 Pascal standard write formats. However, these routines add image formatting to them.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" in the main text.

If the output file is not specified, then the standard **output** file is used.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

procedure writer([var f: text;] r: real; fmt: string);

Convert **real r** to string **s**. Real numbers are well covered by the normal ISO 7185 Pascal standard write formats. However, these routines add image formatting to them.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" in the main text.

If the output file is not specified, then the standard output file is used.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, InvalidRealNumber, InvalidFractionSpecification, NegativeNotPlaced, CurrentBlockFull**

procedure writere([var f: text;] r: real; fl: integer);
procedure writere([var f: text;] r: real);

Write **real r** to the "economy real" format to either the specified output file **f**, or to the standard output. The economy real format is explained in the **reales** routines descriptions.

The versions with a field **fl** fit the output to the field using standard Pascaline rules, including negative fields.

If the output file is not specified, then the standard **output** file is used.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, CurrentBlockFull**

procedure writeh([var f: text;] w: lcardinal [; fl: integer]);
procedure writeh([var f: text;] w: lcardinal; fmt: string);

Write **cardinal w** to either the specified file **f**, or the standard **output** file, using the hexadecimal radix system. The versions with a field **fl** format the number using the field using standard Pascaline rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings".

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NegativeValueNondecimal, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

procedure writeo([var f: text;] w: lcardinal [; fl: integer]);
procedure writeo([var f: text;] w: lcardinal; fmt: string);

Write **cardinal w** to either the specified file **f**, or the standard **output** file, using the octal radix system. The versions with a field **fl** format the number using the field using standard Pascaline rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" above.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NegativeValueNondecimal, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

procedure writeb([var f: text;] w: lcardinal [; fl: integer]);
procedure writeb([var f: text;] w: lcardinal; fmt: string);

Write **cardinal w** to either the specified file **f**, or the standard **output** file, using the binary radix system. The versions with a field **fl** format the number using the field using standard Pascaline rules for output using a field. Negative (left justified) fields are also allowed.

The versions with a **fmt** string copy the format string into the result string, then replace each of the format characters with parts of the number. See I.3 "format strings" above.

Exceptions: **FormatTooLarge, InvalidFieldSpecification, NegativeValueNondecimal, NumberOverflowsFormat, NegativeNotPlaced, CurrentBlockFull**

function intv(s: [p]string [; var ovf: boolean]): integer;

Find value of string **s** with default decimal format. The number contained in the string, in valid Pascal signed number format, is parsed and the value returned. The number can be signed. Leading and trailing spaces are ignored, but extra characters past a valid number result in an error.

If the number is unsigned, and has a valid radix specifier, one of "$", "&" or "%" prepended to it, then that radix overrides the default decimal radix. If this override effect is not desired, then the number should be checked for such radix specifications using **index**, and an error generated.

The versions of this call that feature the **ovf** flag do not produce an error if the number is too large to convert, but instead set the **ovf** flag, and then the result is undefined. Note that this is the same behavior as catching an exception for the non-**ovf** version.

Exceptions: **InvalidRadix, InvalidIntegerFormat, NumberTooLarge, StringNil**

function hexv(s: [p]string [; var ovf: boolean]): lcardinal;

Find value of string **s** with default hexadecimal format. The number contained in the string, in valid Pascal signed number format, is parsed and the value returned. The number may not be signed. Leading and trailing spaces are ignored, but extra characters past a valid number result in an error.

If the number has a valid radix specifier, one of "$", "&" or "%" prepended to it, then that radix overrides the default hexadecimal radix. If this override effect is not desired, then the number should be checked for such radix specifications using index, and an error generated.

The versions of this call that feature the **ovf** flag do not produce an error if the number is too large to convert, but instead set the **ovf** flag, and then the result is undefined (and probably is garbage). Note that this is the same behavior as catching an exception for the non-**ovf** version.

Exceptions: **InvalidRadix, InvalidIntegerFormat, NumberTooLarge, StringNil**

function octv(s: [p]string [; var ovf: boolean]): lcardinal;

Find value of string **s** with default octal format. The number contained in the string, in valid Pascal signed number format, is parsed and the value returned. The number may not be signed. Leading and trailing spaces are ignored, but extra characters past a valid number result in an error.

If the number has a valid radix specifier, one of "$", "&" or "%" prepended to it, then that radix overrides the default octal radix. If this override effect is not desired, then the number should be checked for such radix specifications using index, and an error generated.

The versions of this call that feature the **ovf** flag do not produce an error if the number is too large to convert, but instead set the **ovf** flag, and then the result is undefined (and probably is garbage). Note that this is the same behavior as catching an exception for the non-**ovf** version.

Exceptions: **InvalidRadix, InvalidIntegerFormat, NumberTooLarge, StringNil**

function binv(s: [p]string [; var ovf: boolean]): lcardinal;

Find value of string **s** with default binary format. The number contained in the string, in valid Pascal signed number format, is parsed and the value returned. The number may not be signed. Leading and trailing spaces are ignored, but extra characters past a valid number result in an error.

If the number has a valid radix specifier, one of "$", "&" or "%" prepended to it, then that radix overrides the default binary radix. If this override effect is not desired, then the number should be checked for such radix specifications using index, and an error generated.

The versions of this call that feature the **ovf** flag do not produce an error if the number is too large to convert, but instead set the **ovf** flag, and then the result is undefined (and probably is garbage). Note that this is the same behavior as catching an exception for the non-**ovf** version.

Exceptions: **InvalidRadix, InvalidIntegerFormat, NumberTooLarge, StringNil**

function realv(s: [p]string): real;

Find value of real number string **s**. The string must contain a valid Pascal format real number. Leading and trailing spaces are ignored, but extra characters past a valid number result in an error.

Exceptions: **InvalidIntegerFormat, IntegerTooLarge, InvalidRealFormat, StringNil**

procedure subst[c][all](var s: string; m: string; r: string);
function subst[c][all](s: [p]string; m: [p]string; r: [p]string): pstring;

Substitutes a string within another string. The source string **s** is searched for the match string **m**. If found, the match string is removed, and the replacement string **r** takes its place. The string lengths and positions are readjusted for any size difference between the match and replacement strings. The final string is either returned in the string **s**, or returned as the result.

Several variations exist. The matching process can be case sensitive, or not. In normal **subst** calls, only the first match is replaced. In the "all" variations, all matching substrings are replaced. Replacement strings are not searched in the "all" variations. The match process skips over the replacement string.

Exceptions: **StringNil**

procedure openstring;

Causes a new string block level to be created. All strings that were dynamically allocated within **strings** will be recorded within the new block, and disposed of automatically on **closestring**.

Exceptions: None

procedure closestring;

Removes the current string block level. Each string that was allocated and recorded in the current block is disposed of, and the surrounding string block, if it exists, is restored. All **strings** dynamic allocations will then be recorded in that block.

It is an error if no block exists to close.

Exceptions: **NoStringBlock**

procedure exportstring(s: pstring);

Removes the indicated string **s** completely from the current string block. Only the current block is searched. It is only possible to export a string from the block it was created in. If the string does not exist in the current block, no error results.

Exceptions: **NoStringBlock**

procedure upstring(s: pstring);

Moves the indicated string **s** from the current block to the surrounding block. The string is removed from the current block, and recorded in the surrounding block, if it exists. If the string does not exist in the current block, or a surrounding block does not exist, no error results. If the surrounding block is full, an error results.

Exceptions: **OuterBlockFull**

# J Annex: Extended mathematics library

ISO 7185 Pascal kept the number of implemented transcendental functions limited to simplfy the base implementation. This library introduces several of the more common extensions for math capability, and also defines some special constants used in the IEEE 754 standard for floating point numbers, which is most often used as the basis for computer floating point.

At this writing, IEEE 754 is has virtually taken over as the standard for all hardware implemented floating point math, as well as most software implemented floating point math. However, there is little in this library that cannot be interpreted for other formats as well. In fact, the **nanval** function, that returns the exact code for a given NaN, is the only truly IEEE 754 dependent function in the library. This may be important for legacy implementions that were created before the standard, or for simplified hardware and software implementations using formats different from IEEE 754.

## J.1 Functions

The following functions exist in the **math** library.

| Name | Function |
|---|---|
| **sign(x)** | Find sign x |
| **exp2(x)** | Power function, base 2 x |
| **exp10(x)** | Power function, base 10 x |
| **log2(x)** | Logarithm, base 2 x |
| **log10(x)** | Logarithm, base 10 x |
| **tan(x)** | Tangent x |
| **cot(x)** | Cotangent x |
| **arcsin(x)** | Arc Sine x |
| **arccos(x)** | Arc Cosine x |
| **arccot(x)** | Arc Cotangent x |
| **arctan2(x, y)** | Arc tangent corrected for other quadrants x |
| **pow(x,y)** | x to the power y |
| **sinh(x)** | Hyperbolic Sine x |
| **cosh(x)** | Hyperbolic Cosine x |
| **tanh(x)** | Hyperbolic Tangent x |
| **coth(x)** | Hyperbolic Cotangent x |
| **arcsinh(x)** | Inverse Hyperbolic Sine x |
| **arccosh(x)** | Inverse Hyperbolic Cosine x |
| **arctanh(x)** | Inverse Hyperbolic Tangent x |
| **arccoth(x)** | Inverse Hyperbolic Cotangent x |
| **pinfinity(x)** | Test if positive infinity x |
| **ninfinity(x)** | Test if negative infinity x |
| **nan(x)** | Test if x is a NaN |
| **qnan(x)** | Test if x is a quiet NaN |
| **snan(x)** | Test if x is a signaling NaN |
| **nanval(x)** | Returns the code for a NaN x |
| **frac(x)** | Returns the mantissa, or fractional part of x |
| **expo(x)** | Returns the exponent of x |
| **makereal(e,f)** | Make a real from exponent and fraction e and f |

Unless otherwise specified, all of the functions take and receive **lreal**s. This means that they can both convert from any of **sreal**, **real** or **lreal**, and have their results converted to same.

## J.2 Further trancendentals

**exp2, exp10, log2, log10, tan, cot, arcsin, arcos, arccot, arctan2** and **pow** are provided. Many of these were not included in the ISO 7185 standard because they can be calculated from the other, ISO 7185 functions. They are included here both for convenience and because it's possible they can be calculated with greater precision by a routine specifically created for that function.

## J.3 Hyperbolics

The hyperbolic functions **sinh(x), cosh(x), tanh(x) coth(x), arcsinh(x), arccosh(x), arctanh(x)**, and **arccoth(x)** give the transcendental functions calculated along a hyperbola, instead of a circle.

## J.4 Special floating point values

A number can be tested for positive infinity with **pinfinity(x)**, and negative infinity with **ninfinity(x)**. If the implementation does not discriminate between positive and negative infinity, then all infinities are considered positive.

Predefined constants are available as **pi** and **eu** (for Euler's number). These constants are represented in their maximum precision as **lreal**s.

## J.5 NaN functions

The NaN fuctions return true if the number is the specified NaN. **nan(x)** tests for any class of NaN. **qnan(x)** tests only for quiet, or non-signaling NaNs. **snan(x)** tests for signaling NaNs.

For any NaN value, the function **nanval(x)** returns the exact NaN number. This is the fraction value without the hidden and signal bits.

Note that the Pascaline implementation may treat NaNs as errors, or only return NaNs to the program by special option.

The exact meaning of a particular NaN code is implementation dependent. It is also possible for the NaN codes to have different meanings depending on if a quiet and signaling NaN is indicated.

Note that finding the exact code of a NaN with **nanval** is specific to the IEEE 754 standard.

## J.6 Utility functions

**sign(x)** returns the sign of either real or integer **x** as -1 for negative, and +1 for positive. The function **expo(x)** returns the binary (base 2) exponent of the number **x** as an integer. It is a signed number expressing the number of binary point dividing the whole part of the number from the fractional part, expressed as an offset from the extreme right side of the number.

The function **frac(x)** returns the **lcardinal** value of the mantissa for real number **x**. If the mantissa of **x** has more significant digits than can be contained by **lcardinal**, an error results.

The procedure **makereal(e, f)** expects an exponent value as returned by the **expo** function and a mantissa value as returned by the **frac** function and unites them to make a real floating point result.

## J.7 Exceptions in the math library

The following exceptions are thrown by procedures and functions in **math**.

Note that many exceptions in the math library are the same as standard Pascaline exceptions in Annex O.

| ValueOutOfRange | Value out of range |
|---|---|
| **ZeroDivide** | Zero divide |
| **RealOverflow** | Real overflow. |
| **RealUnderflow** | Real underflow. |
| **RealProcessingFault** | Real processing fault. |
| **InvalidFieldSpecification** | Invalid field specification. |
| **InvalidRealNumber** | Invalid real number. |
| **InvalidFractionSpecification** | Invalid fraction specification; |
| **InvalidRealFormat** | Invalid real format. |
| **ArgumentNegative** | Negative argument is invalid |
| **ArgumentZero** | Zero argument is invalid |
| **ArgumentRange** | Argument out of range |

**math** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **math** will go back to **math**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of math need only capture the exceptions occurring in the procedure or function that is called.

## J.8 Functions, procedures and constants in the math library

function sign(x: lreal): integer;

Find sign of **x**. Returns either a -1 for negative, or +1 for positive.

Exceptions: none

function exp2(x: lreal): integer;

Power function, base 2. Computes 2 to the power **x**, the base 2 exponential of **x**.

Exceptions: none

function exp10(x: lreal): lreal

Power function, base 10. Computes 10 to the power **x**, the base 10 exponential of **x**.

Exceptions: none

function log2(x: lreal): lreal;

Logarithm, base 2. Computes the base 2 logarithm of **x**. If is an error if the **x** is negative or zero.

Exceptions: **ArgumentNegative**, **ArgumentZero**

function log10(x: lreal): lreal;

Logarithm, base 10. Computes the base 10 logarithm of **x**. It is an error if **x** is negative or zero.

Exceptions: **ArgumentNegative**, **ArgumentZero**

function tan(x: lreal): lreal;

Tangent. Finds the tangent of **x**, where **x** is in radians.

Exceptions: **RealOverflow**

function cot(x: lreal): lreal;

Cotangent. Find the cotangent of **x**, where **x** is in radians.

Exceptions: none

function arcsin(x: lreal): lreal;

Arc sine. Finds the arc sine of **x**. **x** must be in the range -1 to 1.

Exceptions: **ArgumentRange**

function arccos(x: lreal): lreal;

Arc Cosine. Finds the arc cosine of **x** in radians. **x** must be in the range -1 to 1.

Exceptions: **ArgumentRange**

function arccot(x: lreal): lreal;

Arc Cotangent. Finds the arc cotangent of **x** in radians. **x** must be in the range -1 to 1.

Exceptions: **ArgumentRange**

function arctan2(x, y: lreal): lreal;

Arc tangent corrected for other quadrants. Computes the value of the arc tangent of **y**/**x** using the signs of both arguments. If is an error if both arguments are zero.

Exceptions: **ArgumentZero**

function pow(x: lreal; y: lineger): lreal;

**x** to the power **y**. Calculates **x** raised to the power **y**. It is an error if **x** is negative.

Exceptions: **ArgumentNegative, ArgumentRange**

function sinh(x: lreal): lreal;

Hyperbolic Sine. Finds the hyperbolic sine of **x**.

Exceptions: **ArgumentRange**

function cosh(x: lreal) : lreal;

Hyperbolic Cosine. Finds the hyperbolic cosine of **x**.

Exceptions: **ArgumentRange**

function tanh(x: lreal) : lreal;

Hyperbolic Tangent. Finds the hyperbolic tangent of **x**.

Exceptions: none

function coth(x: lreal) : lreal;

Hyperbolic Cotangent. Finds the hyperbolic cotangent of **x**.

Exceptions: none

function arcsinh(x: lreal) : lreal;

Inverse Hyperbolic Sine. Finds the hyperbolic arc sine of **x**.

Exceptions: none

function arccosh(x: lreal) : lreal;

Inverse Hyperbolic Cosine. Finds the inverse hyperbolic cosine of **x**. Finds the hyperbolic arc cosine of **x**. **x** must be greater or equal to 1.0.

Exceptions: **ArgumentRange**

function arctanh(x: lreal) : lreal;

Inverse Hyperbolic Tangent. Finds the inverse hyperbolic tangent of **x**. **x** must be less than or equal to 1.0.

Exceptions: **ArgumentRange**

function arccoth(x: lreal) : lreal;

Inverse Hyperbolic Cotangent. Finds the inverse hyperbolic cotangent of **x**

Exceptions: none

function pinfinity(x: lreal): boolean;

Test if positive infinity. Returns true if **x** is positive infinity, otherwise false.

Exceptions: none

function ninfinity(x: lreal): boolean;

Test if negative infinity. Returns true if **x** is negative infinity, otherwise false.

Exceptions: none

function nan(x): boolean;

Test if **x** is a NaN. Returns true if **x** contains a NaN or Not A Number code, either quiet or signaling, otherwise false.

Exceptions: none

function qnan(x: lreal): boolean;

Test if **x** is a quiet NaN. Returns true if **x** contains a quiet NaN, otherwise false.

Exceptions: none

function snan(x: lreal): boolean;

Test if **x** is a signaling NaN. Returns true if **x** contains a signaling NaN, otherwise false.

Exceptions: none

function nanval(x: lreal): lcardinal;

Returns the code for a NaN. Returns the code for the NaN **x**, without the quiet or signaling flag. It is an error if **x** is not a NaN.

Exceptions: NotANaN

function frac(x: lreal): lcardinal;

Returns the mantissa, or fractional part of **x**. The number returned is with the binary point at the far right of the number.

Exceptions: none

function expo(x: lreal): linteger;

Returns the exponent of **x**. The exponent is adjusted so that the mantissa has the binary point at the extreme right of the number, that is, the exponent expresses the number of binary digits to the left or right of that position to represent the number, with positive exponents moving the binary point to the right, and negative exponents moving the binary point to the left.

Exceptions: none

function makereal(e: linteger; f: lcardinal): lreal;

Make a real from exponent and fraction. Accepts an exponent **e** and fraction **f** with the fraction having a binary point at the extreme right side, and the exponent as an offset to that. It is an error if the resulting number cannot be represented in **lreal** format.

Exceptions: **RealOverflow**

# K Annex: Terminal Interface Library

Standard ISO 7185 Pascal uses an I/O paradigm that is serial, or more correctly "line oriented". Each line is built up in sections, then output to the I/O device with an appended "end of line". The end of line causes the current line to be completed, and the next line begins.

The next level of paradigm is the presentation of lines onto a 2d text surface. This can simply be an emulation of the serial only system or “virtual paper” using a screen that scrolls up from the bottom. The next step is to allow full addressing of the 2d surface and allowing text to be placed anywhere within that surface. To this is added colors, different text presentation modes, and finally advanced, multiple device input.

**terminal** starts in ISO 7185 Pascal compatible mode, then allows the program to move to a full addressable surface without automatic scrolling. In advanced mode, **terminal** emulates an infinite virtual surface where the terminal exists as a window clipped to the origin. This is the most consistent model of such a surface.

Because **terminal** overrides the interface between the program and the operating system it runs on, all of the ISO 7185 Pascal defined serial I/O works compatibly to the **terminal** presented surface. It is also modal, meaning that changes to character modes affect all further output to the terminal.

**terminal** introduces the concept that several devices can be used for input at the same time. The normal user keyboard is supplemented by a mouse, timers and a joystick. These are all implemented via the event concept, which unifies the input and removes the need to poll multiple devices.

**terminal** gives a set of logical events for common control keys from the keyboard that allow the program to avoid direct recognition of implementation dependent key codes.

In addition to the default presentation surface, **terminal** is capable of switching between multiple display surfaces. This capability has several uses.

## K.1 ISO 7185 Pascal Compatible Mode

To write data to the **terminal** screen, ISO 7185 Pascal **write** calls are used. A **write** statement places each character on the screen, then moves the cursor to the right, obeying any automatic line wrapping. If a **writeln** is called, then the cursor is returned to the left side of the screen, one line down. If the end of the line is reached, and automatic scrolling is enabled, then the screen will scroll upwards.

The **page** procedure, which causes a printer to move to the next (blank) page, is emulated by waiting for the user to acknowledge the contents of the screen, then the screen is cleared and the cursor moves to the upper left hand position (1,1).

## K.2 Basic Cursor Positioning

The cursor is the point where text is entered onto the screen. It’s usually marked with a blinking block or underline. To move the cursor one character up, down, left or right, the **up**, **down**, **left** and **right** procedures are used. To move the cursor anywhere on the screen, the **cursor** call is used. To find out where the cursor currently is, the functions **curx** and **cury** return the current x and y coordinates of it.

The character cells on the screen are labeled from 1 to N, where in x 1 is the left side of the screen, and N is the right side of the screen. In y 1 is the top of the screen, and N is the bottom of the screen.

The actual size of the screen is system specific, and can be found by **maxx** and **maxy**, which return the maximum index in x and y for the screen. When a terminal is emulated in a windowing system, it usually by default 25 lines of 80 characters each, because that was a very common size in terminals.

The cursor can be moved to the home, or 1,1 position, by the **home** procedure. The entire display can be cleared with the cursor moved to the home position by the **clear** procedure.

## K.3 Automatic Mode

Automatic mode is the mode that causes **terminal** to scroll upwards when the bottom of the screen is reached, and an **eoln** is written. Line wrap, or the wrapping of characters back to the left at the next line if text is written off the right hand side, is an automatic mode. The automatic mode is useful for emulating ISO 7185 Pascal serial mode programs, but rapidly gets in the way for advanced programs.

Automatic mode can be turned off with **auto**. Turning **auto** off converts (or actually, reveals) the screen as a character surface that goes from -**maxint** to +**maxint** in both x and y, and has its origin at 1,1, and the screen is a "viewport" on this surface that extends from 1,1 to **maxx**, **maxy**. Text can be drawn anywhere, including off screen, but the characters outside of the 1,1 to **maxx**, **maxy** box will not be seen, and are "clipped out" of view. It can be determined if the cursor lies within the screen's bounds by **curbnd**.

The "virtual screen" that appears with **auto** off is a good match for today's windowing environments. Text can be drawn without worrying if it will cause the line to wrap, or the screen to scroll. And if a line of text happens to extend off the screen, that does not cause an error.

## K.4 Tabbing

**terminal** will keep track of tabs set in x, for any character position. When **terminal** starts, the tabs are set to every 8th position on the screen, i.e., positions x= 9, 17, etc.

Outputting a tab character will cause the cursor to move to the next tab position on the line. If the cursor is at a tab position, then it will move to the next one. Tab positions can be set by **settab**, and cleared by **restab**. **clrtab** clears all set tabs.

## K.5 Scrolling

The screen can be scrolled by the **scroll** procedure, which implements arbitrary direction scrolling. If the x value given is positive, then the screen data scrolls up. If the x value is negative, then the screen data scrolls down. If the y value is positive, the screen data scrolls left, and if it's negative, the screen data scrolls right. If either x or y is 0, then there is no movement in that direction.

## K.6 Colors

There are two colors to set for text. One is the foreground, and the other is the background. The foreground is the color within the character itself. The background is the space behind the character. The possible colors are chosen from the two sets of primary colors:

```
type color = (black, white, red, green, blue, cyan, yellow,
              magenta);
```

Characters are written in the currently set foreground and background colors. The foreground color is set by **fcolor**, and the background color by **bcolor**.

The current background color is also used to set the color of any blanked out areas caused by other commands. For example, the **clear** procedure clears the screen to the background color. Scrolls, either programmer selected or automatic, use the background color for any uncovered areas that are blanked out.

## K.7 Attributes

**terminal** provides many attribute controls, each of which can be switched on or off individually.

| Attribute | Result |
|---|---|
| **reverse** | Enables reverse video. |
| **underline** | Underlines each character. |
| **superscript** | Gives a smaller and higher character for subscripting. |
| **subscript** | Gives a smaller and lower character for superscripting. |
| **italic** | Prints in italic or slanted characters. |
| **bold** | Prints in extra dark, or bold characters. |
| **Strikeout** | Prints characters with a horizontal bar through them. |
| **blink** | Blinks each character on and off. |

For programs to maintain portability, the programmer should assume two things. First, that only a single attribute at one time can be set. This would mean that setting a second attribute after a first one is set, would cause the first attribute to be unset.

Second, the programmer should not assume that any one particular attribute is available. Many older terminals do have more than one or two attributes. **Superscript**, **subscript**, **italic** and **strikeout** are rare in standalone terminals. **Superscript**, **subscript**, **italic** and **bold** are rare today on graphical windowing systems because they alter the geometry of the characters such that it is difficult or impossible to emulate a character cell in such a system. **blink** is also rare in graphical systems because of the computational work required.

For this reason, there is a general mode, **standout**, that can be enabled or disabled just like an attribute. What **standout** does is enable an attribute the terminal does have, so that the program does not have to select a specific attribute.

## K.8 Multiple Surface Buffering

**terminal** implements the ability to have logical screens in buffers. This is a copy of the characters on the screen, saved in a buffer of the same size as the screen. Logical buffers have many uses. Multiple screens can be kept, and quickly switched to the user. A program, like an editor, that wants to save the screen as it was before it started, can switch away to a second screen to do its work, then switch back to the original screen to restore its contents. Buffers can also be used to accomplish animation.

The select procedure sets both the current screen to be displayed, and also the screen that updates are to go to. When **terminal** starts, these are both set to the same screen, number 1. Any screen can be

selected, and the display screen and the update screen can be different. If the update screen is not the one in display, then all of the write statements will place characters in the buffer, but not on screen. This "split mode" is typically used for animation.

Implementations are free to limit the total number of logical screens available. The user should assume no more than 10 logical screens are available.

## K.9 Advanced Input

**terminal** implements an advanced output model that is upward compatible with the ISO 7185 Pascal serial model. Similarly, an advanced input model is also implemented that meets the needs of newer systems.

Today's deal with multiple input devices, not just the keyboard. Devices include a mouse, a joystick, timers, and other future devices. The standard method on small computers was to poll for such devices, or accept interrupts from them. The difficulty with those solutions was that these methods did not fit well with modern multitasking systems.

One method would be to use a multitask thread per device. However, this complicates small programs unnecessarily. The common solution today is "events", and the "event loop". The event model takes advantage of the fact that user input devices don't generate a lot of high bandwidth data. Each of the devices attached to the input from the user have their data packaged as "messages", which are small records, which are the description of the event. These events are then placed in a queue, and the program pulls the events, one at a time, from the queue and acts on them.

The description of an event record is:

```
module terminal;

const maxtim = 10; { maximum number of timers available }

type

joyhan = 1..4; { joystick handles }
joynum = 0..4; { number of joysticks }
joybut = 1..4; { joystick buttons }
joybtn = 0..4; { joystick number of buttons }
joyaxn = 0..3; { joystick axies }
mounum = 0..4; { number of mice }
mouhan = 1..4; { mouse handles }
moubut = 1..4;  { mouse buttons }
timhan = 1..maxtim; { timer handle }
funky  = 1..100; { function keys }

{ events }

evtcod = (etchar,    { ANSI character returned }
          etup,      { cursor up one line }
          etdown,    { down one line }
          etleft,    { left one character }
          etright,   { right one character }
          etleftw,   { left one word }
          etrightw,  { right one word }
          ethome,    { home of document }
          ethomes,   { home of screen }
          ethomel,   { home of line }
          etend,     { end of document }
          etends,    { end of screen }
          etendl,    { end of line }
          etscrl,    { scroll left one character }
          etscrr,    { scroll right one character }
          etscru,    { scroll up one line }
          etscrd,    { scroll down one line }
          etpagd,    { page down }
          etpagu,    { page up }
          ettab,     { tab }
          etenter,   { enter line }
          etinsert,  { insert block }
          etinsertl, { insert line }
          etinsertt, { insert toggle }
          etdel,     { delete block }
          etdell,    { delete line }
          etdelcf,   { delete character forward }
          etdelcb,   { delete character backward }
          etcopy,    { copy block }
          etcopyl,   { copy line }
```

```
            etcan,     { cancel current operation }
            etstop,    { stop current operation }
            etcont,    { continue current operation }
            etprint,   { print document }
            etprintb,  { print block }
            etprints,  { print screen }
            etfun,     { function key }
            etmenu,    { display menu }
            etmouba,   { mouse button assertion }
            etmoubd,   { mouse button deassertion }
            etmoumov,  { mouse move }
            ettim,     { timer matures }
            etjoyba,   { joystick button assertion }
            etjoybd,   { joystick button deassertion }
            etjoymov,  { joystick move }
            etterm);   { terminate program }

{ event record }

evtrec = record

   case etype: evtcod of { event type }

      { ANSI character returned }
      etchar:   (char:                char);

      { timer handle that matured }
      ettim:    (timnum:              timhan);
      etmoumov: (mmoun:               mouhan;   { mouse number }
                 moupx, moupy:        integer); { mouse movement }
      etmouba:  (amoun:               mouhan;   { mouse handle }
                 amoubn:              moubut);  { button number }
      etmoubd:  (dmoun:               mouhan;   { mouse handle }
                 dmoubn:              moubut);  { button number }
      etjoyba:  (ajoyn:               joyhan;   { joystick number }
                 ajoybn:              joybut);  { button number }
      etjoybd:  (djoyn:               joyhan;   { joystick number }
                 djoybn:              joybut);  { button number }
      etjoymov: (mjoyn:               joyhan;   { joystick number }
                 joypx, joypy, joypz: integer); { joystick
                                                  coordinates }
      etfun:    (fkey:                funky);   { function key }
      etup, etdown, etleft, etright, etleftw, etrightw, ethome,
      ethomes, ethomel, etend, etends, etendl, etscrl, etscrr,
      etscru, etscrd, etpagd, etpagu, ettab, etenter, etinsert,
      etinsertl, etinsertt, etdel, etdell, etdelcf, etdelcb, etcopy,
      etcopyl, etcan, etstop, etcont, etprint, etprintb, etprints,
      etmenu, etterm: (); { normal events }
```

```
   { end }

end;

begin ! terminal
end.
```

The next event for the program is retrieved via the **event** call. The basis of the event system is that events must be retrieved often enough that the input queue does not fill up. Thus, an "event model" program will be centered around the event loop that gets the next event, acts on it, and returns to the top for more events.

An important principle of event handling is that events that are not defined for a compliant program are ignored. That is, if an event not defined for **terminal** is received, it will be ignored. This is the basis of upward compatibility. If a new event is defined, existing programs will simply ignore it.

A typical event loop is as follows.

```
program p;

joins terminal;

var er: evtrec;

begin

   repeat

      terminal.event(er); ! get next event
      case er.etype of

         etchar: if er.char = 'g' then ; ! perform character action
         etmoumov: ! perform mouse move action

         else ! do nothing

      end

   until er.etype = etterm ! until user orders terminate

end.
```

Note the final default case that does nothing.

## K.10 Event callbacks

An alternative to receiving events as records are the event based virtual methods:

```
module terminal;

virtual procedure evchar(c: char); begin end;
virtual procedure evtim(t: timhan); begin end;
virtual procedure evmoumov(m: mouhan); begin end;
virtual procedure evmouba(h: mouhan; b: moubut); begin end;
virtual procedure evmoubd(h: mouhan; b: moubut); begin end;
virtual procedure evjoyba(h: joyhan; b: joybut); begin end;
virtual procedure evjoybd(h: joyhan; b: joybut); begin end;
virtual procedure evjoymov(h: joyhan; x, y, z: integer); begin end;
virtual procedure evfun(k: funky); begin end;
virtual procedure evup; begin end;
virtual procedure evdown; begin end;
virtual procedure evleft; begin end;
virtual procedure evright; begin end;
virtual procedure evleftw; begin end;
virtual procedure evrightw; begin end;
virtual procedure evhome; begin end;
virtual procedure evhomes; begin end;
virtual procedure evhomel; begin end;
virtual procedure evend; begin end;
virtual procedure evends; begin end;
virtual procedure evendl; begin end;
virtual procedure evscrl; begin end;
virtual procedure evscrr; begin end;
virtual procedure evscru; begin end;
virtual procedure evscrd; begin end;
virtual procedure evpagd; begin end;
virtual procedure evpagu; begin end;
virtual procedure evtab; begin end;
virtual procedure eventer; begin end;
virtual procedure evinsert; begin end;
virtual procedure evinsertl; begin end;
virtual procedure evinsertt; begin end;
virtual procedure evdel; begin end;
virtual procedure evdell; begin end;
virtual procedure evdelcf; begin end;
virtual procedure evdelcb; begin end;
virtual procedure evcopy; begin end;
virtual procedure evcopyl; begin end;
virtual procedure evcan; begin end;
virtual procedure evstop; begin end;
virtual procedure evcont; begin end;
virtual procedure evprint; begin end;
virtual procedure evprintb; begin end;
virtual procedure evprints; begin end;
virtual procedure evmenu; begin end;
virtual procedure evterm; begin end;
```

```
begin ! terminal
end.
```

Each event procedure corresponds to an event from the **evtrec** record. When **event** has an event to return to the calling program, it first calls the default implementation of the corresponding virtual procedure, which flags that the event is unhandled, and should be returned to the caller.

The alternative method is activated by overriding the virtual procedure corresponding to the event that is to be received directly. **event** directly calls the overriding procedure, and the event is processed there. If the overriding procedure does not want to handle the event, then the inherited version of the procedure is called in the overrider. This accomplishes two things. First, any other overriders that are chained to the procedure are executed, so that they may handle the event. Finally, if none of the overriders wish to handle the event, the chain ends with the default implementation, which then flags that the event is to be returned to the caller of **event**. In this way, if none of the overriders handle the event, it is returned as a normal record back to the **event** caller.

There is no parallel execution implied in such event callbacks. The **event** function still must be called to activate the event procedures, and all such procedures run in the context of the current process.

The event procedures are prefixed with “ev” (event) to prevent them from colliding with the names of the normal **terminal** procedures.

The event procedures are a way to break the rigid formalism of event loop design. In the event loop model, the event loop must be changed anytime there is new code that needs to handle events. Using event procedures, new handlers can be added without such modification. This enhances modularity, since the new code may be in a different module. This aids the extendibility of the system.

Event procedures are also a way to obfuscate a program by making it less obvious where the flow of control is in the program. The advantage to the event loop model is that it creates clear flow of control, even when handling asynchronous user generated events.

## K.11 Timers

Timers allow a **terminal** program to perform periodic events, such as screen updates, and keeping track of time. From 1 to 10 timers are available, numbered 1..10. Each timer is given a time to measure, in 100 Microsecond counts (see H.3 “Time and Date” in **services** for more details). When the timer is done, it sends an event to the event queue.

Timers can either simply stop when their time is done, or they can automatically start timing their original set time again. This is the "recurrent" mode, and it's useful when an activity needs to be performed periodically for the life of the program. For example, updating the screen on an active program, or checking when to update a clock display.

Timers are set by timer. A timer can be stopped or "killed" by **killtimer**. Killing a timer that is not active will not generate an error. This allows a single run timer to be killed without timing out during the call.

Due to the queuing nature of the event system, there is no guarantee about the accuracy of a timer. It can arrive later than its set time. If the program is late getting back to the event queue, or is busy with other events that occur before the timer, receipt of the timer event can be very late. All that receiving a timer event does is indicate that the time it measured has passed.

An example of timer use is the display of a clock, using the **services** time call. If a recurrent timer is set to go off on every second, the program should ***not*** simply advance the second on each timer event. Instead, the timer event should tell the program to read time to determine if the second has changed, and what value it currently has.

Implementations are free to limit the number of timers. Users should assume no more than 10 timers are available.

## K.12 The Frame Timer

When performing animation, it is common to flip between screen buffers with select. The ideal time to perform a buffer flip is during the retrace time for the display, or the time it takes the display drawing hardware to reset to the top of the screen, and start drawing from the top again. Hiding the buffer flip in the retrace time can be key to making animations appear smooth.

The frame timer is a timer much like the standard timers, except that it is set automatically by the implementation. It is simply enabled or disabled, and gives an **etframe** event when it times out. On hardware that is capable, the **etframe** event is triggered by the beginning of the retrace cycle.

If an interrupt for the start of the retrace cycle is not available, then the frame timer is simply defaulted to a reasonable rate of redraw for animation, for example, 30 times per second.

## K.13 Mouse

The mouse gives a position x,y on the screen, as well as from 1 to 4 buttons on it. A mouse actually gives its position as relative movements in x and y, but the system converts this to a screen position. The function **mouse** returns the number of mice attached to the system.

Mice generate two events. First, when the mouse moves, it generates position changes via the **etmoumov** event. The program does not have to worry about where it is going. Each time the position changes, a new x, y position is posted as an event.

The second event generated by a mouse is mouse button asserts and deasserts, **etmouba** and **etmoubd**. An "assert" means a press of the button, and a "deassert" is the release of the button. Note that instead of an on/off status check as polled by the program, the assert and deassert events give exact notice of when the button changes state, and what it is changing to.

When there is more than one mouse per system, the default behavior should be to treat them as separate, but equal controls on the screen from the same user. When a mouse moves or changes button state, it gets control of the program. This matches the common use of multiple pointing devices, where two devices are alternated by one user. For example, a trackball and a mouse may be just two different input methods from one user.

Alternately, a second mouse could be a remote mouse over a network. This "collaborative computing" model allows two users to look at the same document, with separate mice. Advanced implementations of multiple mice such as these should be selected by the user.

Mice are subject to “rate limiting”. This means that the number of events per second reported by the mouse can be limited to no more than what the human viewer can perceive. This prevents the number of mouse events from affecting program performance.

## K.14 Joysticks

From 0 to 4 joysticks may be supported. Each joystick can have from 0 to 3 "axes" of directions of travel. In addition, each joystick can have 0 to 4 buttons. The function **joystick** returns the number of joysticks in the system. The function **joyaxis** gives the number of axes on a given joystick. The function **joybutton** gives the number of buttons on a given joystick.

The messages **etjoyba** or joystick button assert, and **etjoybd** or joystick button deassert, give events for the assertion and deassertion of the buttons on a joystick. When any axis on a joystick moves, it generates a **etjoymov**, or joystick movement, event. This event gives the relative setting of each axis of the joystick.

Unlike a mouse, a joystick is entirely relative. Each axis is represented by an integer. If the axis is not implemented it always reads 0. If the axis is deflected left, up or in, depending on the type of axis, then it is negative. If it is right, down, or out, it is positive. The axis is determined in its amount of deflection. If it is in the middle, it is 0. If it is deflected to the maximum, it is **maxint**, with the sign giving the direction.

By convention, the axes on a joystick are:

1. Left/right, or slider.
2. Up/down
3. In/out

Joysticks are "rate limited" devices. This means that no matter how fast the joystick is "slewed", it will only produce an event about every 10 milliseconds at maximum rate. The reason for this is that humans cannot generally perceive events like movement of a pointer across the screen at a faster rate than this, so there is no point in updating faster than that, and flooding the input event queue.

## K.15 Function Keys

The system may have function keys, which are keys whose function are determined by the program. The number of function keys implemented are found with **funkey**. Function key messages are sent by the message **etfun**.

## K.16 Automatic “hold” Mode

**terminal** implements a feature to help with legacy programs designed to the ISO 7185 Pascal serial I/O model. When a program exits that is unaware of the terminal model, the terminal window can abruptly close. This means that any printout from the program is lost.

The automatic hold mode keeps the window open unless a **etterm** event is received, until the user specifically closes the window. This mode is enabled by default in systems were it is valuable, i.e., windowing systems.

There are times when the automatic hold mode can be a problem. For example, if a user displayed menu features a “quit” option, it would be incorrect to hold that window after the user has already closed the program.

To solve this, the **autohold** procedure can be used to set automatic hold mode off or on. With automatic hold mode off, the window exits immediately when the program does.

## K.17 Direct Writes

**terminal** accepts all standard ISO 7185 Pascal output methods, **write**, **writeln**, and **page**. However, it can be faster to output characters to the console directly, bypassing the normal file protocol layers inherent in the system. The procedure **wrtstr** outputs a character string directly to the terminal without any interpretation of control characters. It cannot be used with **auto** on.

## K.18 Printers

**Terminal** can be used to operate a printer using a subset of the **terminal** functionality. To model a printer, **terminal** uses a one page buffer that can be written using normal **terminal** commands to write to this "virtual screen" contained within the buffer. When the **page** procedure is executed, the contents of the buffer is output to the printer as a whole page, then a new page can be written.

When **terminal** is connected to a printer file, the following conditions are true:

1. There is no input file associated with the printer, neither event or virtual event methods function. **event** gives an exception. None of the input devices work, and timer, mouse, joystick, function keys, the frame timer and the **autohold** set procedure all give exceptions.
2. The **select** call does not function, and gives an exception.
3. The dimensions given by **maxx** and **maxy** reflect the size of the printed page.
4. The exact set of attributes will be dependent on the printer. **blink**, of course, is never available.

A printer device is viewed as accepting a series of pages to be printed. The last page should be followed by a **page** procedure call, to insure that a partial page is not left in the page buffer, similar to the way ISO 7185 Pascal does not allow partial lines in a text file. **terminal** may automatically add a **page** call if that is not the last operation to the printer.

Printer devices are not associated with other, active screen terminal devices. It is up to the program how to reformat printout to fit the printed page. Specifically, when printing a copy of an onscreen page, extensive reformatting may be required.

Since printer device mode is a subset of **terminal** functionality, it is possible for any program designed to output to a printer to have its output viewed on a terminal screen. Each page will then be presented to the user in turn.

Printer devices are determined by the system, which knows which files are associated with printer devices and which are not. Specifically, the **list** file which appears as a standard Pascaline header file may be a printer device, a terminal, or a normal file.

## K.19 Metafiles

A **terminal** metafile operates identically to a printer file except that all output is written to a file, and not to an external device. The metafile contains not only the standard characters, eoln, page commands, but can also represent the other output functions in **terminal**.

A metafile is opened with the procedure **openmeta(f, x, y)**, where f is the file to open, and x and y are the number of characters in the x and y character grid dimensions. A metafile is closed using the standard Pascaline **close** procedure.

A metafile has the same restrictions on input, event, select and other functions as a printer file does. It does not have an input file or any input functions, nor a **select** procedure.

A metafile contains a complete representation of output to from **terminal**. It should be possible to render a metafile to either a printer or a display, either by directly copying the file to a suitable printer, or by using a conversion utility.

The format of a metafile is not defined. The only requirement is that all of the output operations, including **write**/**writeln** and **page**, are encoded in the file.

## K.20 Remote display

**terminal** is compatible with the use of "remote display" of presentation text. In this mode, the output from **terminal** is encoded and sent over a communications channel. This is done for both input and output functionality, and thus remote display mode is implemented without restriction of functionality (unlike printer and metafile mode). In order for remote display mode to work, the following conditions must be true:

1. All output functions must be applied in the order they were issued by the program.
2. All input events must be received in the order they were generated by the user.

Any synchronization between output and input is up to the program using **terminal**, just as it is in local mode. For example having the user attempt to select a moving target with the mouse would be highly dependent on the overall display rate of the system.

The exact format of the data passing over the communications channel to allow remote mode to function is system dependent.

Terminal is designed to operate efficiently with such remote displays. For example, there is no ability to read characters from the display.

## K.21 Terminal objects

All of the procedures, functions and other declarations in **terminal** are also available in a terminal object of the form:

```
module terminal;

class screen(var input, output: file);

var er: evtrec; ! event record

! Executive methods

procedure cursor(x, y: integer); begin end;
function maxx: integer; begin end;
function maxy: integer; begin end;
procedure home; begin end;
procedure del; begin end;
procedure up; begin end;
procedure down; begin end;
procedure left; begin end;
procedure right; begin end;
procedure blink(e: boolean); begin end;
procedure reverse(e: boolean); begin end;
procedure underline(e: boolean); begin end;
procedure superscript(e: boolean); begin end;
procedure subscript(e: boolean); begin end;
procedure italic(e: boolean); begin end;
procedure bold(e: boolean); begin end;
procedure strikeout(e: boolean); begin end;
procedure standout(e: boolean); begin end;
procedure fcolor(c: color); begin end;
procedure bcolor(c: color); begin end;
procedure auto(e: boolean); begin end;
procedure curvis(e: boolean); begin end;
procedure scroll(x, y: integer); begin end;
function curx: integer; begin end;
function cury: integer; begin end;
function curbnd: boolean; begin end;
procedure select(u, d: integer); begin end;
procedure event; begin end;
procedure timer(i: timhan; t: integer; r: boolean); begin end;
procedure killtimer(i: timhan); begin end;
function mouse: mounum; begin end;
function mousebutton(m: mouhan): moubut; begin end;
function joystick: joynum; begin end;
function joybutton(j: joyhan): joybtn; begin end;
function joyaxis(j: joyhan): joyaxn; begin end;
procedure settab(t: integer); begin end;
procedure restab(t: integer); begin end;
procedure clrtab; begin end;
function funkey: funky; begin end;
function frametimer(e: boolean); begin end;
procedure autohold(e: boolean); begin end;
```

```
procedure wrtstr(s: string); begin end;

! Event callbacks

virtual procedure evchar(c: char); begin end;
virtual procedure evtim(t: timhan); begin end;
virtual procedure evmoumov(m: mouhan); begin end;
virtual procedure evmouba(h: mouhan; b: moubut); begin end;
virtual procedure evmoubd(h: mouhan; b: moubut); begin end;
virtual procedure evjoyba(h: joyhan; b: joybut); begin end;
virtual procedure evjoybd(h: joyhan; b: joybut); begin end;
virtual procedure evjoymov(h: joyhan; x, y, z: integer); begin end;
virtual procedure evfun(k: funky); begin end;
virtual procedure evup; begin end;
virtual procedure evdown; begin end;
virtual procedure evleft; begin end;
virtual procedure evright; begin end;
virtual procedure evleftw; begin end;
virtual procedure evrightw; begin end;
virtual procedure evhome; begin end;
virtual procedure evhomes; begin end;
virtual procedure evhomel; begin end;
virtual procedure evend; begin end;
virtual procedure evends; begin end;
virtual procedure evendl; begin end;
virtual procedure evscrl; begin end;
virtual procedure evscrr; begin end;
virtual procedure evscru; begin end;
virtual procedure evscrd; begin end;
virtual procedure evpagd; begin end;
virtual procedure evpagu; begin end;
virtual procedure evtab; begin end;
virtual procedure eventer; begin end;
virtual procedure evinsert; begin end;
virtual procedure evinsertl; begin end;
virtual procedure evinsertt; begin end;
virtual procedure evdel; begin end;
virtual procedure evdell; begin end;
virtual procedure evdelcf; begin end;
virtual procedure evdelcb; begin end;
virtual procedure evcopy; begin end;
virtual procedure evcopyl; begin end;
virtual procedure evcan; begin end;
virtual procedure evstop; begin end;
virtual procedure evcont; begin end;
virtual procedure evprint; begin end;
virtual procedure evprintb; begin end;
virtual procedure evprints; begin end;
virtual procedure evmenu; begin end;
```

```
virtual procedure evterm; begin end;

.

begin ! terminal
end.
```

The description of each method in a **screen** object appears with the same methods as the module **terminal** in K.23.

When a **screen** object is instantiated, it accepts an input and an output file as parameters. Normally these are the standard **input** and standard **output** files of the program. However, these can be different files in derived objects.

A **screen** object contains an event record, so it is not necessary to specify an external event record in the **event** procedure (nor possible).

A **screen** object can be created as follows:

```
program p;

joins terminal;

var si(input, output): terminal.screen;

begin

   si.home; { send cursor to home position }
   writeln(si.output, 'hello, terminal world');
   repeat { event loop }

      si.event; { get next event }
      { process events }

   until si.er.type = etterm { loop until program cancelled }

end.
```

Where **terminal** is the terminal module, and **si** is the **screen** object. **screen** objects can be instantiated statically or dynamically.

**screen** objects contain their own state for the following:

1. Location of cursor.
2. Current attributes and colors.
3. Buffer working/display status.
4. Tab stops.
5. Timer numbers.

However, each **screen** object gets the complete state of the module **terminal** when it is created.

Having each **screen** object possess its own state means that each object can be writing to its own section of the screen in its own mode. Each object can even have its own private screen buffer, but this must be specifically selected using the **select** procedure.

When multiple **screen** objects exist, and all have the cursor visibility on, the cursor will remain at the last place it was specifically directed to. This can mean that the cursor will rapidly appear to swap back and forth between the active drawing **screen** objects. This can be improved by only selecting one of the **screen** objects as having a visible cursor. Typically, a the cursor should be placed to attract user attention to where text is being entered, or if none is being entered, it should be turned off.

As in the procedural interface to **terminal**, events in the **screen** class can be registered as callbacks via the virtual procedures. However, just as in the procedural interface, such callbacks do not function unless the **event** method for the **screen** object is called.

## K.22 Exceptions

The following exceptions are generated in **terminal**:

| Identifier | Meaning |
|---|---|
| **TooManyFiles** | The total number of open files possible was exceeded. |
| **NoJoyStick** | No joystick access was available. |
| **NoTimer** | No timer access was available. |
| **ToManyTimers** | The total number of timers available was exceeded. |
| **FilenameEmpty** | Filename specified was empty. |
| **InvalidScreenNumber** | Screen number specified was invalid. |
| **InvalidHandle** | An invalid handle was specified. |
| **InvalidTab** | An invalid tab position was specified. |
| **CannotCreateScreenBuffer** | Cannot create a buffer for the screen. |
| **CannotQueryJoystick** | Could not get information on joystick. |
| **InvalidJoystickHandle** | Invalid handle specified for joystick. |
| **InvalidTimerHandle** | Invalid handle specified for timer. |
| **CannotWriteDirect** | Cannot write direct string with **auto** on. |

**terminal** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **terminal** will go back to **terminal**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **terminal** need only capture the exceptions occurring in the procedure or function that is called.

Besides the exceptions on terminal procedures, the terminal module also can throw new exceptions on existing standard calls **read**, **readln**, **write**, **writeln**, **page**, **reset**, **rewrite**, **assign**, **close**, **length**, **location**, **position**, **update**, **append**, **exists**, **change** and **delete**, all of which are overridden by **terminal**.

## K.23 Procedures, functions and methods in terminal

For all of the following module calls, If the screen file **f** is not present, the default is the standard **output** file, except for **event**, which defaults to the standard **input** file. If the procedure or function is a method, neither the input nor output screen file can be specified, since it is inherent in the object.

procedure cursor([var f: text;] x, y: integer);

Set cursor location for output surface file **f** in **x** and **y**.

Exceptions: None

function maxx[(var f: text)]: integer;

Find maximum screen location **x** in output surface file **f**.

Exceptions: None

function maxy[(var f: text)]: integer;

Find maximum screen location **y** in output surface file **f**.

Exceptions: None

procedure home[(var f: text)];

Send cursor to 1,1 location (upper left of screen) in output surface file **f**.

Exceptions: None

procedure clear[(var f: text)];

Clears the screen to the current background color and sends the cursor to 1,1 location (upper left of screen) in output surface file **f**.

Exceptions: None

procedure del[(var f: text)];

Back up cursor by one character in output surface file **f**, and erase character at that location. If the cursor is at the left side of the screen, and automatic mode is on, the cursor will be moved up one line, and to the right of screen. If the cursor is at the top of the screen, extreme left, then the screen will be scrolled down one line, and the cursor moves to the right side of the screen.

Exceptions: None

procedure up[(var f: text)];

Move cursor up one line in output surface file **f**. If the cursor is already at the top line, and automatic mode is on, the screen will be scrolled down, and the cursor remains at the same position.

Exceptions: None

procedure down[(var f: text)];

Move cursor down one line in output surface file **f**. If the cursor is already at the bottom line, and automatic mode is on, the screen will be scrolled up, and the cursor remains at the same position.

Exceptions: None

procedure left[(var f: text)];

Back up cursor by one character in output surface file **f**. If the cursor is at the left side of the screen, and automatic mode is on, the cursor will be moved up one line, and to the right of screen. If the cursor is at the top of the screen, left, then the screen will be scrolled down one line, and the cursor moves to the right side of the screen.

Exceptions: None

procedure right[(var f: text)];

Move forward by one character in output surface file **f**. If the cursor is at the right side of the screen, and automatic mode is on, the cursor will be moved down one line, and to the left of screen. If the cursor is at the bottom of the screen, right, then the screen will be scrolled up one line, and the cursor moves to the left side of the screen.

Exceptions: None

procedure blink([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in blinking text in output surface file **f**.

Exceptions: None

procedure reverse([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in reverse text in output surface file **f**.

Exceptions: None

procedure underline([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in underlined text in output surface file **f**.

Exceptions: None

procedure superscript([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in superscript text in output surface file **f**.

Exceptions: None

procedure subscript([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in subscript text in output surface file **f**.

Exceptions: None

procedure italic([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in italic text in output surface file **f**.

Exceptions: None

procedure bold([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in bold text in output surface file **f**.

Exceptions: None

procedure strikeout([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in strikeout text in output surface file **f**.

Exceptions: None

procedure standout([var f: text;] e: boolean);

If **e** is true, causes all further characters written to the screen to appear in standout text in output surface file **f**. Standout is assigned to the first mode possible from the following order:

Reverse.
Underline.
Bold
Italic.
Strikeout.
Blink.

If none of those modes are available, **standout** is a no-op.

Exceptions: None

procedure fcolor([var f: text;] c: color);

Sets the foreground, or text color, to the color **c** in output surface file **f**.

Exceptions: None

procedure bcolor([var f: text;] c: color);

Sets the background, or space color, to the color **c** in output surface file **f**.

Exceptions: None

procedure auto([var f: text;] e: boolean);

Turns automatic mode on or off, according to **e** in output surface file **f**.

Exceptions: None

procedure curvis([var f: text;] e: boolean);

Turns cursor visibility on or off, according to **e** in output surface file **f**.

Exceptions: None

procedure scroll([var f: text;] x, y: integer);

Scroll in arbitrary directions. The screen is scrolled according to the differences in **x** and **y**. Uncovered areas on the screen appear in the current background color.

Exceptions: None

function curx[(var f: text)]: integer;

Find the current **x** location of the cursor in output surface file **f**.

Exceptions: None

function cury[(var f: text)]: integer;

Find the current **y** location of the cursor in output surface file **f**.

Exceptions: None

function curbnd[(var f: text)]: boolean;

Check cursor in bounds. Returns true if the cursor is currently within the bounds of the screen in output surface file **f**.

Exceptions: None

procedure select([var f: text;] u, d: integer);

Select buffer to update and display. Selects the active buffer for update **u**, and display **d** in output surface file **f**. The update buffer will receive the result of all writes. The display buffer will be shown on screen.

Exceptions: **CannotCreateScreenBuffer**

procedure event[([var f: text;] var er: evtrec)];

Get next event. Retrieves the next event from the input queue from the terminal input file **f** to event record **er**. If there is no event ready, the program will wait.

Note that specification of the event record is optional only for the terminal class method.

Exceptions: None

procedure timer([var f: text;] i: timhan; t: integer; r: boolean);

Set timer active in output surface file **f**. The timer **i** will be set to run for time **t**. If the repeat flag **r** is set, then the timer will automatically repeat when the time expires. If the timer is already in use, then it will cease its current timing, and perform the new time.

Exceptions: **InvalidHandle**

procedure killtimer([var f: text;] i: timhan);

Stop timer in output surface file **f**. Stops the timer **i**. If the timer is not active, no error is reported.

Exceptions: **InvalidHandle**

function mouse[(var f: text)]: mounum;

Returns the number of mice in output surface file **f**.

Exceptions: None

function mousebutton([var f: text;] m: mouhan): moubut;

Returns the number of buttons on a given mouse **m** in output surface file **f**.

Exceptions: None

function joystick[(var f: text)]: joynum;

Returns the number of joysticks in the system in output surface file **f**.

Exceptions: None

function joybutton([var f: text;] j: joyhan): joybtn;

Returns the number of buttons on the given joystick **j** in output surface file **f**.

Exceptions: None

function joyaxis([var f: text;] j: joyhan): joyaxn;

Returns the number of axes on the given joystick **j** in output surface file **f**.

Exceptions: None

procedure settab([var f: text;] t: integer);

Set new tab. Sets a new tab location at **t** in output surface file **f**.

Exceptions: **InvalidTab**

procedure restab([var f: text;] t: integer);

Reset tab. Removes the tab at location **t** in output surface file **f**. If there is not a tab set there, it is not an error.

Exceptions: **InvalidTab**

procedure clrtab[(var f: text)];

Clear all tabs. All tabs are removed from the tabbing table in output surface file **f**.

Exceptions: None

function funkey[(var f: text)]: funky;

Returns the number of function keys available in output surface file **f**.

Exceptions: None

function frametimer([var f: text;] e: boolean);

Enables or disables the framing timer in output surface file **f**. If **e** is true, the frame timer is enabled, otherwise disabled. The frame timer gives frame timer events, which occur approximately on each refresh of the display screen. It may be tied to the refresh hardware, or may be simulated via a timer.

Exceptions: **NoTimer**

procedure autohold([var f: text;] e: boolean);

Sets the state of automatic hold in output surface file **f**. If **e** is true, **autohold** is enabled, otherwise disabled. **autohold** determines if the window will exit immediately if the program self terminates. If an exit was not ordered via the user interface, the display is held until it is.

Exceptions: None

procedure wrtstr([var f: text;] s: string);

Writes the string **s** directly to the output surface file **f**. No control character interpretation is done. This procedure is used to perform efficient writes to the display surface without per-character overhead.

It is an error to call this routine when **auto** is enabled.

Exceptions: **CannotWriteDirect**

## K.24 Events and Callbacks In terminal

For each item, both the event record section and the virtual procedure is presented. See the description of the event record (K.9 “Advanced Input”) for the format of the entire record.

Event: etchar: (c: char);
virtual procedure evchar(c: char);

Returns a keyboard character **c**.

Event: ettim: (timnum: timhan);
virtual procedure evtim(t: timhan);

Indicates the timer according to the timer handle **t** has expired.

Event: etmoumov
virtual procedure evmoumov(h: mouhan; x, y: integer);

The mouse with handle **h** has moved, to the position indicated by **x** and **y**.

Event: etmouba
virtual procedure evmouba(h: mouhan; b: moubut);

The mouse with handle **h** asserted the button **b**.

Event: etmoubd
virtual procedure evmoubd(h: mouhan; b: moubut);

The mouse with handle **h** deasserted the button **b**.

Event: etjoyba
virtual procedure evjoyba(h: joyhan; b: joybut);

The joystick with handle **h** asserted the button **b**.

Event: etjoybd
virtual procedure evjoybd(h: joyhan; b: joybut);

The joystick with handle **h** asserted the button **b**.

Event: etjoymov
virtual procedure evjoymov(h: joyhan; x, y, z: integer);

The joystick with handle **h** moved, and the coordinates **x**, **y** and **z**. The values of each axis are between –**maxint**..**maxint**. The number of axis actually present in the given joystick are given by the function **joyaxis**. The value returned by an unimplemented axis is undefined.

Event: etfun
virtual procedure evfun(k: funky);

A function key was sent from the keyboard, with **k** giving the number of the key.

Event: etup
virtual procedure evup;

The key for move cursor up was sent from the keyboard.

Event: etdown
virtual procedure evdown;

The key for move cursor down was sent from the keyboard.

Event: etleft
virtual procedure evleft;

The key for move cursor left was sent from the keyboard.

Event: etright
virtual procedure evright;

The key for move cursor right was sent from the keyboard.

Event: etleftw
virtual procedure evleftw;

The key for move cursor left word was sent from the keyboard. This indicates the cursor should be moved left one “word”, or over any series of non-space characters.

Event: etrightw
virtual procedure evrightw;

The key for move cursor right word was sent from the keyboard. This indicates the cursor should be moved right one “word”, or over any series of non-space characters.

Event: ethome
virtual procedure evhome;

The key for move cursor to the home position in the document (top extreme left) was sent from the keyboard.

Event: ethomes
virtual procedure evhomes;

The key for move cursor to the home position in the screen (top extreme left) was sent from the keyboard.

Event: ethomel
virtual procedure evhomel;

The key for move cursor to the home position in the line (extreme left) was sent from the keyboard.

Event: etend
virtual procedure evend;

The key for move cursor to the end position in the document (bottom extreme right) was sent from the keyboard.

Event: etends
virtual procedure evends;

The key for move cursor to the end position in the screen (bottom extreme right) was sent from the keyboard.

Event: etendl
virtual procedure evendl;

The key for move cursor to the end position in the line (extreme right) was sent from the keyboard.

Event: etscrl
virtual procedure evscrl;

The key for scroll screen left one character was sent from the keyboard.

Event: etscrr
virtual procedure evscrr;

The key for scroll screen right one character was sent from the keyboard.

Event: etscru
virtual procedure evscru;

The key for scroll screen up one character was sent from the keyboard.

Event: etscrd
virtual procedure evscrd;

The key for scroll screen down one character was sent from the keyboard.

Event: etpagd
virtual procedure evpagd;

The key for page down was sent from the keyboard.

Event: etpagu
virtual procedure evpagu;

The key for page up was sent from the keyboard.

Event: ettab
virtual procedure evtab;

The key for enter tab was sent from the keyboard.

Event: etenter
virtual procedure eventer;

The key for enter line was sent from the keyboard.

Event: etinsert
virtual procedure evinsert;

The key for insert block was sent from the keyboard.

Event: etinsertl
virtual procedure evinsertl;

The key for insert line was sent from the keyboard.

Event: etinsertt
virtual procedure evinsertt;

The key for insert toggle was sent from the keyboard. The action should be to toggle the state of the insert/overwrite flag used to determine if new typed text overwrites previous text on screen, or inserts new text between existing characters.

Event: etdel
virtual procedure evdel;

The key for delete block was sent from the keyboard.

Event: etdell
virtual procedure evdell;

The key for delete line was sent from the keyboard.

Event: etdelcf
virtual procedure evdelcf;

The key for delete character forward was sent from the keyboard. This indicates the character to the right of the cursor should be deleted.

Event: etdelcb
virtual procedure evdelcb;

The key for delete character backward was sent from the keyboard. This indicates the character to the left of the cursor should be deleted.

Event: etcopy
virtual procedure evcopy;

The key for copy block was sent from the keyboard. This indicates the currently selected block should be copied.

Event: etcopyl
virtual procedure evcopyl;

The key for copy line was sent from the keyboard. This indicates the current line should be copied.

Event: etcan
virtual procedure evcan;

The key for cancel current operation was sent from the keyboard. This indicates the operation in progress should be canceled.

Event: etstop
virtual procedure evstop;

The key for stop current operation was sent from the keyboard. This indicates the operation in progress should be stopped.

Event: etcont
virtual procedure evcont;

The key for continue current operation was sent from the keyboard. This indicates the operation in progress should be continued if stopped previously.

Event: etprint
virtual procedure evprint;

The key for print current document was sent from the keyboard. This indicates the current document should be printed in entirety.

Event: etprintb
virtual procedure evprintb;

The key for print current block was sent from the keyboard. This indicates the current selected bock, if it exists, should be printed.

Event: etprints
virtual procedure evprints;

The key for print current screen was sent from the keyboard. This indicates the current screen should be printed.

Event: etmenu
virtual procedure evmenu;

The key for display menu was sent from the keyboard. This indicates the menu, if any, should be displayed.

Event: etterm
virtual procedure evterm;

The key for terminate program was sent from the keyboard. This indicates the program should be exited.

If this event is received, then the user ordered the exit. This means that automatic hold mode, if enabled, will be bypassed, and the program closed immediately.

# L Annex: Graphical Interface Library

The graphical library **graphics** extends the **terminal** model by adding graphical output procedures.

Since it is completely upward compatible with **terminal**, and standard ISO 7185 Pascal serial output modes, any program from ISO 7185 Pascal, or **terminal** compliant Pascaline will run under **graphics**. In the most advanced modes of **graphics**, ordinary Pascal **write** statements can still be used to output text, so all of the output formatting procedures of ISO 7185 Pascal still work.

**graphics** will handle any graphics task. Besides drawing features, it supports double page animation. Combined with the sound library **sound**, full graphical games are possible.

## L.1 Terminal model

**graphics** emulates **terminal** by setting up a "character grid" across the pixel based screen. Each "cell" on the grid matches the pixel height and width of a character. The character font is set to a fixed font by default when **graphics** starts. This gives every character in the font the same height and width. Finally, the text drawing mode is set such that both the foreground (the inside of the characters) and the background (the space behind the characters) are drawn, overwriting any content of each character cell as it is written.

This mode can be kept, while drawing other figures on the same text surface. Alternately, the cursor can be set to any arbitrary pixel on the screen, and text written anywhere, down to the pixel. Also, the font can be changed to a proportional one for a more pleasing look. Because automatic mode relies on characters being neatly placed on the grid, it must be turned off before the cursor leaves the grid or becomes a proportional font.

When **auto** is turned off, the grid is still useful. Although the character spacing varies, the line spacing is still valid. This means that the grid is no longer useful in the x direction, but it is still useful in the y direction. In addition, the origin in x is still valid. The result is that a series of lines printed with end-of-lines will do the right thing with **auto** off, namely present a series of left justified lines at the x origin (flush left) with the correct spacing.

## L.2 Graphics Coordinates

The total size of the graphics screen is found by **maxxg** and **maxyg**, which return the maximum pixel index in x and y. The pixel coordinates on the screen are from 1,1 to **maxxg(f)**, **maxyg(f)**. The cursor can be set to any pixel position by **cursorg(f, x, y)**. The current location of the cursor in pixel terms is found by **curxg** and **curyg**.

## L.3 Character Drawing

There can be any number of fonts available on the system, including both fixed space fonts, and proportional fonts that vary in the width of characters. The number of fonts in the system can be found with the **font** function. Fonts are chosen by logical number with **font(f, c)**, where **c** is the font code, 1 to **font**. There are two methods to determine what font is assigned to a particular font code. The first is the standard font codes, the second is the font name system. Standard fonts are numbers for commonly used fonts in the system. These are commonly available fonts the program may need.

Font 1: Terminal Font

> This is the default font set up by **graphics** when it starts. It's a fixed font. It also cannot be superscripted, subscripted, bold or italic, because these modes change the size of the font.

Font 2: Book Font

> This is a serif font, and is good for general purpose text such as what a paragraph in a book is written in. This is the most common proportional font.

Font 3: Sign Font

> This is a no serif font (sans serif), and is best for headings, titles and similar uses, as in road signs and other signs. It's a proportional font.

Font 4: Technical Font

> The technical font is a fixed font that is guaranteed to be able to scale to any arbitrary size. This font is used to label drawings and engineering documents. Before the technology existed to create arbitrary fonts in any point size, the technical font was done with stroked vector graphics. Now, it is more likely to be equivalent to the sign font.

Beyond the standard fonts, the name of an installed font can be found by **fontnam(f, fc, fns)**, where **fc** is the font code, 1 to **font**, and **fns** returns the descriptive string for the font, such as "Helvetica". The application should use the standard fonts by default, then present the system fonts, by name to the user, and let the user choose one of them.

The size of each character is set by **fontsiz(f, n)**, where n is the height of the font in pixels. The reason character sizes are set by their height is because proportional fonts vary in the width of the character. The height never varies. The current height of the font is found by **chrsizy**. Its width is found with **chrsizx**. When a proportional font is active, **chrsizx** returns the width of a space in that font, which is always as wide or wider than the widest character in that font.

Besides the basic font, extra space can be added between lines (known as "leading" in typography, for the lead strips used between type lines) with **chrspcy(f, n)**. n is the number of pixels of extra space to add between lines. Extra space between characters is added with **chrspcx(f, n)**, where n is the number of pixels of extra space to add between characters.

The graphical cursor is placed at the upper left of the character box for the next character to be drawn. If it is needed to know exactly where the character will rest if drawn on a line. The offset from the top of the character box to the baseline of the text is found with the function **baseline**.

In typography, fonts and characters are measured by points, of which there are 28.35 points per centimeter. Point measurement implies that the exact size of objects drawn on the screen is known. This can be determined by the functions **dpmx** and **dpmy**, which return the "dots per meter" or pixels in one meter for both x and y. The reason it can be two different measures for the two different axes is that the display may not have square pixels or a 1:1 aspect ratio.

To find a given point size in terms of the height needed for the character, it is found by:

dpmy/2835*point size

The screen aspect ratio can also be found from these calls, it is:

```
dpmx/dpmy
```

## L.4 String Sizes and Kerning

When writing text to the screen in proportional font, it is often needed to know exactly how much space the string will take up on the screen, down to the pixel. This amount of space can change not only because of the variable width of proportional fonts, but also because of an effect called "kerning". When a string of characters is draw together, the system can apply "kerning" to characters in the string. Kerning means to fit the individual letters together, like puzzle pieces, to come up with a tighter spacing than is normally possible. For example, "A" and "V" together as "AV" can typically be kerned together to take less space because they can overlap. To find the exact number of pixels in x that a string will occupy, the function **strsiz(f, s)** is used, where s is the string. The size of the string in y does not change, and can be found with **chrsizy**. To find the exact position, in pixels offset from the beginning of the string in x, of a given character, use **chrpos(f, s, n)**, where s is the string, and n is the index of the character to find the x offset of.

## L.5 Justification

Justification is the spreading of spacing through a string of characters to fit a given space. If the string will fit into the space is found with **strsiz(f, s)**, and checking if the resulting pixels required are less than or equal to the space they will occupy as justified. The character string is written in justified mode with **writejust(f, s, n)**, where **s** is the string to write, and **n** is the number of pixels to fit it in. If the number of pixels allowed for is not enough, the string will be larger than the requested number. The offset, in x, of a given justified character, is found with **justpos(f, s, p, n)**, where **s** is the string, **p** is the offset of the character you are interested in, and **n** is the total number of pixels to fit the string in, as in **writejust**.

## L.6 Effects

**graphics** expands the effects in **terminal**. For smaller character baselines, **condensed(f, b**), is used. For larger character baselines, **extended(f, b)** is used**.**

In addition to normal **bold**, there are also **light(f, b)**, **xlight(f, b)**, and **xbold(f, b)** effects. For lighter than normal, extra light, and extra bold modes.

Characters will have an embossed look with **hollow(f, b)** and **raised(f, b)**. Hollow makes the character look sunken, and raised makes it look as if coming off the page.

## L.7 Tabs

**graphics** extends the character level of tabbing in **terminal** with procedures that can set tabs on an individual pixel. **settabg(f, x)** sets a tab at the pixel **x**. **restabg(f, x)** resets the tab at pixel **x**. The **terminal** procedure **clrtab(f)** clears all tabs, including pixel level tabs.

## L.8 Colors

The simple eight colors from **terminal** are still available, with the addition of two new calls that allow access to the full range of colors an advanced graphic system provides. **fcolorg(f, r, g, b)** sets the foreground color from values of red **r**, green **g**, and blue **b**. Similarly, **bcolorg(f, r, g, b)** sets the background color from rgb values. The values of the colors are ratioed. This means that instead of an

absolute number, the possible colors are ratioed from 0 to **maxint**, where 0 is dark, and **maxint** is saturated color. Color ratios allow the true color range implemented by the system to be hidden.

## L.9 Drawing Modes

When colors, background or foreground, are drawn on the screen, they can be in a number of mixing modes. Mixing modes govern how the new color is laid over the old. The modes are mutually exclusive, so the setting of a new mix mode for a given color deactivates the old mode. If the old color is simply to be overwritten, then **fover(f)** or **bover(f)** is appropriate. If the new color is to be xor'ed with the old color, then **fxor(f)** or **bxor(f)** is used. If the new color is to be ignored, leaving the old color underneath intact, use **finvis(f)** or **binvis(f).** There is also **fand(f), band(f), for(f) and bor(f).**

There might not seem to be a use for an "invisible" color, but there are actually several uses. First, if the background is set invisible, the text can be overlaid on another pattern. In fact, this is the most common drawing mode, and most programmers will prefer to turn the background off and lay the backgrounds themselves. Similarly, leaving the background on, then setting the foreground invisible can be used to "stencil" letters with arbitrary patterns inside the letters.

Xor mode is good for several things. First, if a series of figures, say rectangles, are to be laid, but the intersections between them are to be left visible, xor is the right mode. Second, xor can be used to place, and then remove a pattern, even a complex one, easily. This is used to allow the user to place figures by dragging them with the mouse to a new location. It can be used to draw "rubber band" boxes around selections.

Xor mode has two rules of interest:

1. Any two colors, even the same colors, xored together, will give a third color, with the exception of black.
2. Having xored a drawing into the viewplane, xoring the same color and drawing into the viewplane again will restore the old drawing.

Xor can be used for several special effects. However, Xor does not tolerate inaccuracy. Xoring something back off the screen has to be done the same way it was put on, with the same parameters. Also, the mode of drawing in **graphics** is not compatible with some uses of the xor mode. Drawing a rectangle, for example, will result in "corner errors", because the rectangles are built from lines, and lines start and end at the coordinates of the box corners. Because one line starts at the same point another ends, they will xor mix together. The result is a recognizable point of off-color on each corner.

The best way to use xor is to xor a single figure onto the screen, then xor it back off. If complex figures are to be xored on and off the screen, xor is run backwards, that is, from the last figure drawn back to the first.

And and or modes are used to create stencils and other effects. For example, a drawing and’ed with a black stencil will remain black in the stencil area, and intact elsewhere.

## L.10 Drawing Graphics

A graphics element that is not a character is referred to as a "figure". What **graphics** tries to do is provide a small toolset, that does not include figures that you could reasonably construct from the lower level figures. For example, there is no circle figure in **graphics**, because that is simply a special case of **ellipse**.

A parameter that applies to almost all figures is the width of lines. The width of a line usually defaults to 1, but may be more if a single line is unusable on the current display. This can easily happen on a very high resolution display. The line width can be set by the procedure **linewidth(f, n)**, where **n** is the number of pixels for the line to use. There is no limit on the width of a line, and in fact, lines are a defacto way to draw arbitrary angle filled rectangles.

When the line width is set to an even number of pixels, an effect called "even line uncertainty" exists. If you draw a line between two points, and the line width is say, 2, one of the 2 pixels is going to be on the line, and the other could be to either side of it. It literally depends on how the math happens to round off.

To prevent this, line width should always be set to an odd number.

## L.11 Figures

The fundamental figure in graphics is the line. A line is drawn, in the current **linewidth**, by **line(f, x1, y1, x2, y2)**. A rectangle is drawn with **rect(f, x1, y1, x2, y2)**, whose borders have the current **linewidth**. A filled rectangle is drawn with **frect(f, x1, y1, x2, y2)**, whose interior is the foreground color. An ellipse is drawn with **ellipse(f, x1, y1, x2, y2)**. The x and y parameters define a rectangle that contains the figure. The procedure **fellipse(f, x1, y1, x2, y2)** draws a filled ellipse.

The procedure **arc(f, x1, y1, x2, y2, rs, re)** draws an arc line around the ellipse formed by the rectangle formed by the x and y parameters. The start and end points of the arc are described by a special ratio notation that gives the angle. The angles in a 360 degree circle are described by a number from 0 to **maxint**. 0 is the 0 degree, or top center, of the ellipse. The angles around the circle clockwise then go from 0 to **maxint**, at which time a full 360 degrees have been traversed. For example, **maxint** div 2 is 180 degrees, **maxint** div 4 is 90 degrees, etc. The parameter **rs** gives the angle where the arc starts. The parameter **re** gives the angle where the arc ends. Arcs can be specified to cross **maxint** back to zero, or use negative degrees, or any combination.

The procedure **farc(f, x1, y1, x2, y2, rs, re)** draws a filled arc. The procedure **fcord(f, x1, y1, x2, y2, rs, re)** draws a filled cord (a line bisecting the circle).

Rectangles with rounded corners can be drawn with **rrect(f, x1, y1, x2, y2, xw, yw)**. The x and y parameters describe the bounding box, The **xw** and **yw** parameters describe the size of the ellipses that are placed in the corners to round the edges of the box. To draw a filled rounded rectangle, use **frrect(f, x1, y1, x2, y2, xw, yw)**.

The general purpose shape **ftriangle(f, x1, y2, x2, y2, x3, y3)** draws a filled triangle. Parting with convention, **graphics** does not give complex polygon procedures. Rather, you can build up polygons from triangles, and in any case, a high speed drawing engine in hardware would accept triangles only, so the lower level software would have to break up the polygon for you.

Single pixels can be set with **setpixel(f, x, y)**.

## L.12 Predefined Pictures

A picture, or a bitmap, is defined outside the program by a drawing application. Its format is typically operating system specific. **graphics** considers pictures to be a cached resource. A picture is loaded from a file by **loadpict(f, p, fs)**. The string **fs** indicates the file name for the picture file. **p** is a

logical picture number, from 1 to n, and indicates how you want to refer to the picture while its loaded into memory.

A logical picture is drawn onto the screen with **picture(f, p, x1, y1, x2, y2)**. The parameter **p** indicates the logical picture to draw. The x and y parameters indicate the box that the picture is to be drawn into. **graphics** will scale or stretch the picture as needed to make the picture fit into the space given.

In order to determine the parameters of a picture, such as native size and aspect, the functions **pictsizx(f, p)** and **pictsizy(f, p)** are used. These give the native size of the picture in x and y, and the aspect ratio of the picture is then found with **pictsizex(f, p)**/**pictsizey(f, p)**.

### L.13 Scrolling

As in **terminal**, **graphics** can scroll in arbitrary directions. It can also scroll down to the pixel, using the call **scrollg(f, x, y)**. The parameters work the same way as the character position parameters of scroll, except that pixels are specified instead of characters.

### L.14 Clipping

Clipping is completely automatic in **graphics**. Any figure drawn is clipped to the edges of the screen. If a figure is drawn entirely outside the screen bounds, it is completely clipped out.

### L.15 Mouse Graphical Position

A new event, **etmoumovg**, exists that gives mouse movements in pixels, not just characters. The old **etmoumov** still occurs, and carries the character grid message. The **etmoumovg** message happens when the mouse moves a pixel, and the **etmoumov** message happens when the mouse moves a whole character cell. If you don't need the **etmoumov** message, you simply ignore it.

### L.16 Animation

In **terminal**, the **select** call was introduced, that switches between multiple screen buffers. This call is tailor made for double buffer animation, it works in **graphics** the same way. Double buffer animation works by having two screen buffers. One of them is drawn, and the other one is displayed. When a "frame" is drawn, i.e., the picture in the buffer currently being draw is complete, the drawing and display buffers are swapped, and then drawing begins on what used to be the display buffer, clearing it if necessary.

Double buffering removes any of the ongoing drawing operations from the active screen. Although computers do this quite fast, seeing drawing operations in progress tends to produce annoying flashing effects, sparkles and other effects during the drawing. In addition, although the worst interactions with the painting of the graphics card memory to the display screen have disappeared, there can still be odd effects from the fact that the display is being refreshed across the image being drawn.

**graphics** supports more than double buffering (triple, quad or better). However, the advantages of this typically diminish as the amount of data being managed grows without a compensating gain in drawing speeds.

To reduce flicker effects the buffer is flipped when the display enters its retrace period. Syncing with the display eliminates the effects that occur when writing to the display while the display is drawing across it. The retrace period is when the refresh cycle has finished the last lines on the screen, and will

head back to the top of the screen. This gives a short time while the display is not doing anything, so if the buffer swap can occur within the retrace, no effects will appear on the screen at all.

The solution is the **etframe** message. **etframe** is sent when the display enters refresh. If the system does not allow notification for retrace, the **etframe** message is generated by a timer that keeps either the screen refresh rate, or 30 cycles per second if that cannot be determined.

### L.17 Copy between buffers

Blocks of pixels can be copied between buffers with procedure **blockcopyg(f, s, d, sx1, sy1, sx2, sy2, dx1, dy1, dx2, dy2)**. This copies a pixel block from a source bounding box to a destination box in the same or a different buffer. It is capable of both resizing the block, as well as using the write mode to place the pixels.

Copying between buffers is a very powerful technique to speed animation. Pictures can be constructed and processed in a non-displayed buffer, then copied quickly to a target buffer for display. A program with a lot of animation will typically have an offline buffer just to keep sprites and other drawn objects ready to present. The drawing mode can be used to create stencils and other drawing tools.

### L.18 Printers

As with **terminal**, **graphics** can output to a printer if the printer is graphics capable. The one page buffer contains sufficient pixels to allow a complete page with graphics to be rendered in the buffer, then output to the printer with a **page** operation.

See **terminal** for a list of restrictions on printer operation.

### L.19 Metafiles

Metafiles can be written in **graphics**. The same comments that apply to metafiles in **terminal** apply to metafiles in **graphics**.

### L.20 Remote display

**Graphics** is capable of using a remote display as in **terminal**. The same comments as in **terminal** apply to **graphics**.

### L.21 Declarations

The declarations for **graphics** are very similar to **terminal**, with the addition of the mouse move graphical event, and the standard font codes.

```
module graphics;

const

maxtim = 10; { maximum number of timers available }
maxbuf = 10; { maximum number of buffers available }
font_term = 1; { terminal font }
font_book = 2; { book font }
font_sign = 3; { sign font }
font_tech = 4; { technical font (vector font) }

type

{ colors displayable in text mode }
color = (black, white, red, green, blue, cyan, yellow, magenta);
joyhan = 1..4; { joystick handles }
joynum = 0..4; { number of joysticks }
joybut = 1..4; { joystick buttons }
joybtn = 0..4; { joystick number of buttons }
joyaxn = 0..3; { joystick axies }
mounum = 0..4; { number of mice }
mouhan = 1..4; { mouse handles }
moubut = 1..4;  { mouse buttons }
timhan = 1..maxtim; { timer handle }
funky  = 1..100; { function keys }

{ events }

evtcod = (etchar,     { ANSI character returned }
          etup,       { cursor up one line }
          etdown,     { down one line }
          etleft,     { left one character }
          etright,    { right one character }
          etleftw,    { left one word }
          etrightw,   { right one word }
          ethome,     { home of document }
          ethomes,    { home of screen }
          ethomel,    { home of line }
          etend,      { end of document }
          etends,     { end of screen }
          etendl,     { end of line }
          etscrl,     { scroll left one character }
          etscrr,     { scroll right one character }
          etscru,     { scroll up one line }
          etscrd,     { scroll down one line }
          etpagd,     { page down }
          etpagu,     { page up }
          ettab,      { tab }
          etenter,    { enter line }
```

```
            etinsert,   { insert block }
            etinsertl,  { insert line }
            etinsertt,  { insert toggle }
            etdel,      { delete block }
            etdell,     { delete line }
            etdelcf,    { delete character forward }
            etdelcb,    { delete character backward }
            etcopy,     { copy block }
            etcopyl,    { copy line }
            etcan,      { cancel current operation }
            etstop,     { stop current operation }
            etcont,     { continue current operation }
            etprint,    { print document }
            etprintb,   { print block }
            etprints,   { print screen }
            etfun,      { function key }
            etmenu,     { display menu }
            etmouba,    { mouse button assertion }
            etmoubd,    { mouse button deassertion }
            etmoumov,   { mouse move }
            ettim,      { timer matures }
            etjoyba,    { joystick button assertion }
            etjoybd,    { joystick button deassertion }
            etjoymov,   { joystick move }
            etterm,     { terminate program }
            etmoumovg,  { mouse move graphical }
            etframe);   { frame sync }

{ event record }

evtrec = record

   case etype: evtcod of { event type }

      { ANSI character returned }
      etchar:   (char:                char);
      { timer handle that matured }
      ettim:     (timnum:              timhan);
      etmoumov:  (mmoun:               mouhan;   { mouse number }
                  moupx, moupy:        integer); { mouse movement }
      etmouba:   (amoun:               mouhan;   { mouse handle }
                  amoubn:              moubut);  { button number }
      etmoubd:   (dmoun:               mouhan;   { mouse handle }
                  dmoubn:              moubut);  { button number }
      etjoyba:   (ajoyn:               joyhan;   { joystick number }
                  ajoybn:              joybut);  { button number }
      etjoybd:   (djoyn:               joyhan;   { joystick number }
                  djoybn:              joybut);  { button number }
      etjoymov:  (mjoyn:               joyhan;   { joystick number }
```

```
                joypx, joypy, joypz: integer); { joystick
                                                 coordinates }
      etfun:     (fkey:                funky);   { function key }
      etmoumovg: (mmoung:              mouhan;   { mouse number }
                  moupxg, moupyg:      integer); { mouse movement }
      etup, etdown, etleft, etright, etleftw, etrightw, ethome,
      ethomes, ethomel, etend, etends, etendl, etscrl, etscrr,
      etscru, etscrd, etpagd, etpagu, ettab, etenter, etinsert,
      etinsertl, etinsertt, etdel, etdell, etdelcf, etdelcb, etcopy,
      etcopyl, etcan, etstop, etcont, etprint, etprintb, etprints,
      etmenu, etterm, etframe: (); { normal events }

   { end }

end;

begin ! graphics
end.
```

Note that the name of the graphical terminal module retains the name of the original terminal module, but has graphical functionality added.

## L.22 Event callbacks

As in **terminal**, the events can also be accessed via a series of **virtual** procedures. Only one new event procedure exists in the **graphics** module over the original event procedures in **terminal**:

```
module graphics;

virtual procedure evmoumov(h: mouhan; x, y: integer); begin end;

begin ! graphics
end.
```

## L.23 Graphical Terminal Objects

All of the procedures, functions and other declarations in **graphics** are also available in a class of the form:

```
module graphics;

class surface(var input, output: file);

extends screen;

var er: evtrec; ! event record

! Executive methods

procedure cursor(x, y: integer); begin end;
function maxx: integer; begin end;
function maxy: integer; begin end;
procedure home; begin end;
procedure del; begin end;
procedure up; begin end;
procedure down; begin end;
procedure left; begin end;
procedure right; begin end;
procedure blink(e: boolean); begin end;
procedure reverse(e: boolean); begin end;
procedure underline(e: boolean); begin end;
procedure superscript(e: boolean); begin end;
procedure subscript(e: boolean); begin end;
procedure italic(e: boolean); begin end;
procedure bold(e: boolean); begin end;
procedure strikeout(e: boolean); begin end;
procedure standout(e: boolean); begin end;
procedure fcolor(c: color); begin end;
procedure bcolor(c: color); begin end;
procedure auto(e: boolean); begin end;
procedure curvis(e: boolean); begin end;
procedure scroll(x, y: integer); begin end;
function curx: integer; begin end;
function cury: integer; begin end;
function curbnd: boolean; begin end;
procedure select(u, d: integer); begin end;
procedure event; begin end;
procedure timer(i: timhan; t: integer; r: boolean); begin end;
procedure killtimer(i: timhan); begin end;
function mouse: mounum; begin end;
function mousebutton(m: mouhan): moubut; begin end;
function joystick: joynum; begin end;
function joybutton(j: joyhan): joybtn; begin end;
function joyaxis(j: joyhan): joyaxn; begin end;
procedure settab(t: integer); begin end;
procedure restab(t: integer); begin end;
procedure clrtab; begin end;
function funkey: funky; begin end;
```

```
function frametimer(e: boolean); begin end;
procedure autohold(e: boolean); begin end;
procedure wrtstr(s: string); begin end;
function maxxg: integer; begin end;
function maxyg: integer; begin end;
function curxg: integer; begin end;
function curyg: integer; begin end;
procedure line(x1, y1, x2, y2: integer); begin end;
overload procedure line(var f: text; x2, y2: integer); begin end;
overload procedure line(x2, y2: integer); begin end;
overload procedure linewidth(w: integer); begin end;
procedure rect(x1, y1, x2, y2: integer); begin end;
procedure frect(x1, y1, x2, y2: integer); begin end;
procedure rrect(x1, y1, x2, y2, xs, ys: integer); begin end;
procedure frrect(x1, y1, x2, y2, xs, ys: integer); begin end;
procedure ellipse(x1, y1, x2, y2: integer); begin end;
procedure fellipse(x1, y1, x2, y2: integer); begin end;
procedure arc(x1, y1, x2, y2, sa, ea: integer); begin end;
procedure farc(x1, y1, x2, y2, sa, ea: integer); begin end;
procedure fchord(x1, y1, x2, y2, sa, ea: integer); begin end;
procedure ftriangle(x1, y1, x2, y2, x3, y3: integer); begin end;
overload procedure ftriangle(var f: text; x2, y2, x3, y3: integer);
begin end;
overload procedure ftriangle(x2, y2, x3, y3: integer); begin end;
overload procedure ftriangle(var f: text; x3, y3: integer); begin
end;
overload procedure ftriangle(x3, y3: integer); begin end;
procedure cursorg(x, y: integer); begin end;
function baseline: integer; begin end;
procedure setpixel(x, y: integer); begin end;
procedure fover; begin end;
procedure bover; begin end;
procedure finvis; begin end;
procedure binvis; begin end;
procedure fxor; begin end;
procedure bxor; begin end;
function chrsizx: integer; begin end;
function chrsizy: integer; begin end;
function fonts: integer; begin end;
procedure font(fc: integer); begin end;
procedure fontnam(fc: integer; var fns: string); begin end;
procedure fontsiz(s: integer); begin end;
procedure chrspcy(s: integer); begin end;
procedure chrspcx(s: integer); begin end;
function dpmx: integer; begin end;
function dpmy: integer; begin end;
function strsiz(view s: string): integer; begin end;
function strsizp(view s: string): integer; begin end;
function chrpos(view s: string; p: integer): integer; begin end;
```

```
procedure writejust(view s: string; n: integer); begin end;
function justpos(view s: string; p, n: integer): integer; begin end;
procedure condensed(e: boolean); begin end;
procedure extended(e: boolean); begin end;
procedure xlight(e: boolean); begin end;
procedure light(e: boolean); begin end;
procedure xbold(e: boolean); begin end;
procedure hollow(e: boolean); begin end;
procedure raised(e: boolean); begin end;
procedure settabg(t: integer); begin end;
procedure restabg(t: integer); begin end;
procedure fcolorg(var f: text; r, g, b: integer); begin end;
procedure fcolor(var f: text; r, g, b: integer); begin end;
overload procedure fcolor(r, g, b: integer); begin end;
procedure bcolorg(r, g, b: integer); begin end;
procedure bcolor(var f: text; r, g, b: integer); begin end;
overload procedure bcolor(r, g, b: integer); begin end;
procedure loadpict(p: integer; view fn: string); begin end;
function pictsizx(p: integer): integer; begin end;
function pictsizy(p: integer): integer; begin end;
procedure picture(p: integer; x1, y1, x2, y2: integer); begin end;
procedure delpict(p: integer); begin end;
procedure scrollg(x, y: integer); begin end;

! Event callbacks

virtual procedure evchar(c: char); begin end;
virtual procedure evtim(t: timhan); begin end;
virtual procedure evmoumov(m: mouhan); begin end;
virtual procedure evmouba(h: mouhan; b: moubut); begin end;
virtual procedure evmoubd(h: mouhan; b: moubut); begin end;
virtual procedure evjoyba(h: joyhan; b: joybut); begin end;
virtual procedure evjoybd(h: joyhan; b: joybut); begin end;
virtual procedure evjoymov(h: joyhan; x, y, z: integer); begin end;
virtual procedure evfun(k: funky); begin end;
virtual procedure evup; begin end;
virtual procedure evdown; begin end;
virtual procedure evleft; begin end;
virtual procedure evright; begin end;
virtual procedure evleftw; begin end;
virtual procedure evrightw; begin end;
virtual procedure evhome; begin end;
virtual procedure evhomes; begin end;
virtual procedure evhomel; begin end;
virtual procedure evend; begin end;
virtual procedure evends; begin end;
virtual procedure evendl; begin end;
virtual procedure evscrl; begin end;
virtual procedure evscrr; begin end;
```

```
virtual procedure evscru; begin end;
virtual procedure evscrd; begin end;
virtual procedure evpagd; begin end;
virtual procedure evpagu; begin end;
virtual procedure evtab; begin end;
virtual procedure eventer; begin end;
virtual procedure evinsert; begin end;
virtual procedure evinsertl; begin end;
virtual procedure evinsertt; begin end;
virtual procedure evdel; begin end;
virtual procedure evdell; begin end;
virtual procedure evdelcf; begin end;
virtual procedure evdelcb; begin end;
virtual procedure evcopy; begin end;
virtual procedure evcopyl; begin end;
virtual procedure evcan; begin end;
virtual procedure evstop; begin end;
virtual procedure evcont; begin end;
virtual procedure evprint; begin end;
virtual procedure evprintb; begin end;
virtual procedure evprints; begin end;
virtual procedure evmenu; begin end;
virtual procedure evterm; begin end;
virtual procedure evmoumovg(m: mouhan); begin end;
virtual procedure evframe; begin end;

begin ! surface

end.

begin ! graphics

end.
```

The complete catalog of procedures and functions from **graphics** are available as methods in the **surface** class. There are minor differences, which are detailed with the procedure and function descriptions in K.23.

The class **screen** as specified in **terminal** is carried forward into graphics and extended to become **surface**.

When a **surface** object is instantiated, it accepts an input and an output file as parameters. Normally these are the standard **input** and standard **output** files of the program. However, this can be different files in derived objects.

A **surface** object also contains an event record, so it is not necessary to specify an external event record in the **event** procedure.

A **surface** object can be created as follows:

```
program p;

joins graphics;

var si(input, output): graphics.surface;

begin

   si.home; { send cursor to home position }
   writeln(si.output, 'hello, terminal world');
   repeat { event loop }

      si.event; { get next event }
      { process events }

   until si.er.type = etterm { loop until program cancelled }

end.
```

Where **si** is the **surface** object. **surface** objects can be instantiated statically or dynamically.

**surface** objects contain their own state for the following:

1. Location of cursor.
2. Current attributes and colors.
3. Current font and size.
4. Buffer working/display status.
5. Tab stops.
6. Timer numbers.
7. Cached pictures.

However, each **surface** object gets the complete state of the module **graphics** when it is created.

Having each **surface** object possess its own state means that each object can be writing to its own section of the screen in its own mode. Each object can even have its own private screen buffer, but this must be specifically selected using the **select** procedure.

When multiple **surface** objects exist, and all have the cursor visibility on, the cursor will remain at the last place it was specifically directed to. This can mean that the cursor will rapidly appear to swap back and forth between the active drawing screen objects. This can be improved by only selecting one of the screen objects as having a visible cursor. Typically, a the cursor should be placed to attract user attention to where text is being entered, or if none is being entered, it should be turned off.

## L.24 Exceptions

The following exceptions are generated in **graphics**:

| Identifier | Meaning |
|---|---|
| **TooManyFiles** | The total number of open files possible was exceeded. |
| **NoJoyStick** | No joystick access was available. |
| **NoTimer** | No timer access was available. |
| **ToManyTimers** | The total number of timers available was exceeded. |
| **CannotPerformSpecial** | Cannot perform operation on special file. A system file cannot be positioned, etc. |
| **FilenameEmpty** | Filename specified was empty. |
| **InvalidScreenNumber** | Screen number specified was invalid. |
| **InvalidHandle** | An invalid handle was specified. |
| **InvalidTab** | An invalid tab position was specified. |
| **CannotCreateScreenBuffer** | Cannot create a buffer for the screen. |
| **CannotQueryJoystick** | Could not get information on joystick. |
| **InvalidJoystickHandle** | Invalid handle specified for joystick. |
| **InvalidTimerHandle** | Invalid handle specified for timer. |
| **CannotWriteDirect** | Cannot write direct string with **auto** on. |
| **CannotPositionTextByPixel** | Cannot position text by pixel with **auto** on. |
| **CannotPositionOutsideScreen** | Cannot position outside screen with **auto** on. |
| **CannotReenableAutoOffGrid** | Cannot re-enable **auto** off grid |
| **CannotReenableAutoOutsideScreen** | Cannot re-enable **auto** outside screen. |
| **InvalidFontNumber** | Invalid font number. |
| **NoValidTerminalFont** | Valid terminal font not found. |
| **CannotResizeFontWithAuto** | Cannot resize a font with **auto** enabled. |
| **CannotChangeFontsWithAuto** | Cannot change fonts with **auto** enabled. |
| **InvalidFontNumber** | Invalid font number. |
| **NoFontForNumber** | Logical font number has no font assigned. |
| **CannotSizeTerminalFont** | Cannot change the size of the terminal font. |
| **TooManyTabs** | Too many tabs are set. |
| **CannotTabGraphicalWithAuto** | Cannot set a graphical tab with **auto** enabled. |
| **StringIndexOutOfRange** | String index out of range. |
| **PictureFileNotFound** | No picture file was found by filename. |
| **PictureFilenameToLarge** | The specified picture filename was too large. |
| **CannotJustifySystemFont** | Cannot justify system font. |

**graphics** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **graphics** will go back to **graphics**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **graphics** need only capture the exceptions occurring in the procedure or function that is called.

## L.25 Procedures and functions in graphics

See **terminal** for the basic text functions. These are all implemented in **graphics**.

For all of the following calls, If the screen file **f** is not present, the default is the standard **output** or standard **input** file. If the procedure or function is a method, neither the **input** nor **output** screen file should be specified, since it is inherent in the object.

function maxxg[(var f: text)]: integer;

Returns the maximum pixel index x in output surface file **f**.

Exceptions: None

function maxyg[(var f: text)]: integer;

Returns the maximum pixel index y in output surface file **f**.

Exceptions: None

function curxg[(var f: text)]: integer;

Returns the current location of the cursor, in pixel units, for x in output surface file **f**.

Exceptions: None

function curyg[(var f: text)]: integer;

Returns the current location of the cursor, in pixel units, for y in output surface file **f**.

Exceptions: None

procedure line([var f: text;] x1, y1, x2, y2: integer);

Draw line. Draws a line between the point **x1**,**y1** to **x2**,**y2**, using the current line width, color and mode in output surface file **f**.

Exceptions: None

procedure linewidth([var f: text;] w: integer);

Set line width. Sets the line drawing width at **w** pixels wide in output surface file **f**. Use of an odd number of pixels is recommended.

Exceptions: None

procedure rect([var f: text;] x1, y1, x2, y2: integer);

Draw rectangle. Draws a rectangle whose opposite corners are **x1**,**y1** and **x2**,**y2**. Uses the current line width, color and mode in output surface file **f**.

Exceptions: None

procedure frect([var f: text;] x1, y1, x2, y2: integer);

Draw filled rectangle. Draws a solid rectangle, whose opposite corners are **x1**,**y1** and **x2**,**y2**. Uses the current color and mode.

Exceptions: None

procedure rrect([var f: text;] x1, y1, x2, y2, xs, ys: integer);

Draw rounded rectangle. Draws a rectangle with rounded corners. The opposite corners of the bounding box are specified as **x1**,**y1** to **x2**,**y2**. The ellipses that specify the rounded corners are **xs** and **ys**, which specify the width and height of the ellipse. Uses the current line width, color and mode.

Exceptions: None

procedure frrect([var f: text;] x1, y1, x2, y2, xs, ys: integer);

Draw filled rounded rectangle. Draws a rectangle with rounded corners. The opposite corners of the bounding box are specified as **x1**,**y1** to **x2**,**y2**. The ellipses that specify the rounded corners are **xs** and **ys**, which specify the width and height of the ellipse. Uses the current color and mode.

Exceptions: None

procedure ellipse([var f: text;] x1, y1, x2, y2: integer);

Draw an ellipse. Draws an ellipse bonded by a box whose opposite corners are **x1**,**y1** to **x2**,**y2**. Uses the current line width, color and mode.

Exceptions: None

procedure fellipse([var f: text;] x1, y1, x2, y2: integer);

Draw a filled ellipse. Draws a solid ellipse bonded by a box whose opposite corners are **x1**,**y1** to **x2**,**y2**. Uses the current color and mode.

Exceptions: None

procedure arc([var f: text;] x1, y1, x2, y2, sa, ea: integer);

Draw an arc. Draws an arc on an ellipse, whose bounding box is **x1**,**y1** to **x2**,**y2**. The arc starts on the ellipse from the angle **sa**, to the angle **ea**. The angles are given in 360 degree to **maxint** ratio form. Uses the current color and mode.

Exceptions: None

procedure farc([var f: text;] x1, y1, x2, y2, sa, ea: integer);

Draw a filled arc. Draws a filled arc on an ellipse, whose bounding box is **x1**,**y1** to **x2**,**y2**. The arc starts on the ellipse from the angle **sa**, to the angle **ea**. The angles are given in 360 degree to **maxint** ratio form. Uses the current color and mode.

Exceptions: None

procedure fchord([var f: text;] x1, y1, x2, y2, sa, ea: integer);

Draw a filled cord. Draws a filled cord on an ellipse, whose bounding box is **x1**,**y1** to **x2**,**y2**. The cord starts on the ellipse from the angle **sa**, to the angle **ea**. The angles are given in 360 degree to **maxint** ratio form. Uses the current color and mode.

Exceptions: None

procedure ftriangle([var f: text;] x1, y1, x2, y2, x3, y3: integer);

Draw a filled triangle. Draws a filled triangle given the three points **x1**,**y1**,**x2**,**y2**, and **x3**,**y3**. Uses the current color and mode.

Exceptions: None

procedure cursorg([var f: text;] x, y: integer);

Position the cursor graphically. Moves the cursor to the pixel position **x** and **y**.

Exceptions: **CannotPositionTextByPixel**

function baseline[(var f: text)]: integer;

Find baseline of current font. Finds the baseline, or line on which all characters of the current font sit upon, in terms of offset from the character bounding box origin in y.

Exceptions: None

procedure setpixel([var f: text;] x, y: integer);

Set single pixel. Sets a single pixel at the point **x**,**y**, using the current color and mode.

Exceptions: None

procedure fover[(var f: text)];

Set overwrite mode foreground. Sets all new drawing to overwrite old colors on the foreground.

Exceptions: None

procedure bover[(var f: text)];

Set overwrite mode background. Sets all new drawing to overwrite old colors on the background.

Exceptions: None

procedure finvis[(var f: text)];

Set invisible mode foreground. Sets all new drawing to discard colors on the foreground.

Exceptions: None

procedure binvis[(var f: text)];

Set invisible mode background. Sets all new drawing to discard colors on the background.

Exceptions: None

procedure fxor[(var f: text)];

Set xor mode foreground. Sets all new drawing to xor old colors with new colors on the foreground.

Exceptions: None

procedure bxor[(var f: text)];

Set xor mode background. Sets all new drawing to xor old colors with new colors on the background.

Exceptions: None

procedure fand[(var f: text)];

Set and mode foreground. Sets all new drawing to and old colors with new colors on the foreground.

Exceptions: None

procedure band[(var f: text)];

Set and mode background. Sets all new drawing to and old colors with new colors on the background.

Exceptions: None

procedure for[(var f: text)];

Set or mode foreground. Sets all new drawing to or old colors with new colors on the foreground.

Exceptions: None

procedure bor[(var f: text)];

Set or mode background. Sets all new drawing to or old colors with new colors on the background.

Exceptions: None

function chrsizx[(var f: text)]: integer;

Find character size in x. Returns the x size, in pixels, of the characters in the current font. If the font is proportional, its x size will vary per character. The size will then be the space character, which is guaranteed to be the widest character in the font.

Exceptions: None

function chrsizy[(var f: text)]: integer;

Find character size in y. Returns the y size, in pixels, of the characters in the current font.

Exceptions: None

function fonts[(var f: text)]: integer;

Find number of fonts. Returns the number of fonts installed on the system.

Exceptions: None

procedure font([var f: text;] fc: integer);

Select logical font. Selects a font by logical number **fc**, where **fc** is 1..fonts.

Exceptions: **CannotChangeFontsWithAuto**

procedure fontnam([var f: text;] fc: integer; var fns: string);

Find name of logical font. Returns the name of the logical font **fc** in the string **fns**.

Exceptions: **InvalidFontNumber**

procedure fontsiz([var f: text;] s: integer);

Set font size. Sets the height of the current font, in pixels.

Exceptions: **CannotResizeFontWithAuto, CannotSizeTerminalFont**

procedure chrspcy([var f: text;] s: integer);

Set character y spacing. Sets the line to line spacing to s, in pixels.

Exceptions: None

procedure chrspcx([var f: text;] s: integer);

Set character x spacing. Sets the character to character spacing **s**, in pixels.

Exceptions: None

function dpmx[(var f: text)]: integer;

Find dots per meter x. Finds the dots per meter in x of the current display device.

Exceptions: None

function dpmy[(var f: text)]: integer;

Find dots per meter y. Finds the dots per meter in y of the current display device.

Exceptions: None

function strsiz([var f: text;] s: string): integer;

Find pixel size of string. Finds the total x size of the string **s**, in pixels. Accounts for all sizes and spacing.

Exceptions: None

function chrpos([var f: text;] s: string; p: integer): integer;

Find pixel offset of character. Finds the pixel offset from the start of a string **s** in terms of x pixels, to the character by the index **p**.

Exceptions: **StringIndexOutOfRange**

procedure writejust([var f: text;] s: string; n: integer);

Write string justified. Writes the given string s into the number of x pixels n. If the space is more than is required, the extra space will be distributed between the characters. If the space given is insufficient for the characters to be drawn, it will be drawn in the minimum amount of space for the entire string.

Exceptions: **CannotPositionTextByPixel**, **CannotJustifySystemFont**

function justpos([var f: text;] s: string; p, n: integer): integer;

Find justified pixel off set of character. Finds the pixel offset from the start of a string **s**, in terms of x pixels, to the character by the index **p**, for a string justified to fit into **n** pixels. The rules of justification are the same as for **writejust**.

Exceptions: **StringIndexOutOfRange**

procedure condensed([var f: text;] e: boolean);

Set condensed mode. Sets the current font to occupy a shorter baseline than normal. Note that this effect may not be implemented.

Exceptions: None

procedure extended([var f: text;] e: boolean);

Set extended mode. Sets the current font to occupy a longer baseline than normal. Note that this effect may not be implemented.

Exceptions: None

procedure xlight([var f: text;] e: boolean);

Set extra light mode. Sets the current font to extra light printing. Note that this effect may not be implemented.

Exceptions: None

procedure light([var f: text;] e: boolean);

Set light mode. Sets the current font to light printing. Note that this effect may not be implemented.

Exceptions: None

procedure xbold([var f: text;] e: boolean);

Set extra bold mode. Sets the current font to extra bold printing. Note that this effect may not be implemented.

Exceptions: None

procedure hollow([var f: text;] e: boolean);

Set hollow mode. Sets the current font to hollow, or sunken look, printing. Note that this effect may not be implemented.

Exceptions: None

procedure raised([var f: text;] e: boolean);.

Set raised mode. Sets the current font to raised, or relief look, printing. Note that this effect may not be implemented.

Exceptions: None

procedure settabg([var f: text;] t: integer);

Set graphical tab. Sets a tab to the pixel **t**.

Exceptions: **TooManyTabs**

procedure restabg([var f: text;] t: integer);

Reset graphical tab. The tab at pixel **t** is removed. If no tab is set at **t**, no error will result.

Exceptions: **InvalidTab**

procedure fcolorg([var f: text;] r, g, b: integer);

Set foreground color graphical. The foreground color is set to the red **r**, green **g** and blue **b** color. The colors are in 0..**maxint** ratios, with 0 = black, and **maxint** = saturated. The nearest color to the one given is found and set active as the foreground color. The exact color that results will depend on the total number of colors implemented in the system, and could well be black and white in a system so equipped.

Exceptions: None

procedure bcolorg([var f: text;] r, g, b: integer);

Set background color graphical. The background color is set to the red **r**, green **g** and blue **b** color. The colors are in 0..**maxint** ratios, with 0 = black, and **maxint** = saturated. The nearest color to the one given is found and set active as the foreground color. The exact color that results will depend on the total number of colors implemented in the system, and could well be black and white in a system so equipped.

Exceptions: None

procedure loadpict([var f: text;] p: integer; fn: string);

Load picture. Loads the picture from the filename **fn** to logical picture number **p**, which is 1..n. The file must be in a format that the system understands, and is converted to an in memory format that is optimal for the system, such as a direct match for the graphics device in use.

Exceptions: **InvalidHandle, PictureFilenameToLarge**

function pictsizx([var f: text;] p: integer): integer;

Find picture size x. Returns the size, in pixels of x, of the logical picture **p**.

Exceptions: **InvalidHandle**

function pictsizy([var f: text;] p: integer): integer;

Find picture size y. Returns the size, in pixels of y, of the logical picture **p**.

Exceptions: **InvalidHandle**

procedure picture([var f: text;] p: integer; x1, y1, x2, y2: integer);

Draw picture. The logical picture **p** is drawn into the bounding box formed by **x1**,**y1** to **x2**,**y2**. The picture is stretched or compressed as required to fix into the destination bounding box. The method used to stretch or compress may depend on how much computing power is available on the system. There is no attention paid to aspect ratio. If the aspect ratio is to be preserved, it must be calculated. The current foreground mode is used.

Exceptions: **InvalidHandle**

procedure delpict([var f: text;] p: integer);

Remove logical picture from use. The logical picture **p** is removed from the picture queue, and will no longer take up memory space.

Exceptions: **InvalidHandle**

procedure scrollg([var f: text;] x, y: integer);

Scroll in arbitrary directions. The screen is scrolled according to the differences in **x** and **y**, which are in pixels. Uncovered areas on the screen appear in the current background color.

Exceptions: None

procedure blockcopyg([var f: text;] s, d, sx1, sy1, sx2, sy2, dx1, dy1, dx2, dy2 : integer);

Copy a pixel block between buffers. Copies a block of pixels from the source buffer s with the bounding box sx1, sy1 to sx2, sy2 to the destination buffer d in bounding box dx1, dy1 to dx2, dy2. The current foreground write mode is used. The picture is stretched or compressed as required to fit into the destination bounding box. The method used to stretch or compress may depend on how much computing power is available on the system. There is no attention paid to aspect ratio. If the aspect ratio is to be preserved, it must be calculated. The current foreground mode is used.

Scroll in arbitrary directions. The screen is scrolled according to the differences in **x** and **y**, which are in pixels. Uncovered areas on the screen appear in the current background color.

Exceptions: None

## L.26 Events and Callbacks In graphics

For each item, both the event record section and the virtual procedure is presented. See the description of the event record (L.21 “Declarations”) for the format of the entire record.

Event: etmoumovg
**virtual procedure** evmoumovg(h: mouhan; x, y: integer);

The mouse with handle **h** has moved, to the graphical position indicated by **x** and **y**.

# M Annex: Windows Management Library

**windows** is completely upward compatible with **terminal** and **graphics**. Given a single fixed screen, **windows** subdivides the screen into virtual windows, which can be set to any size or position. A **terminal** compatible program sees its window as a "virtual screen", and does not know that some or all of the contents of that screen may be hidden. A **windows** aware program can participate in the benefits of a windowed environment.

## M.1 Screen Appearance

The idea of windows management is to take a program that thinks it is talking to an ordinary terminal, and allow the presentation of multiple such windows within a single screen.

By default, **windows** emulates a **terminal** interface for each window that is identical to the behavior of a full screen window from the programs' point of view. A standard size screen is implemented for the program of 80 characters by 25 lines (even if the real display has a different size). **windows** accepts program writes, and places that information in a buffer that looks like an ideal screen. Then, the contents of that buffer are placed on the screen in various arrangements to complete the desktop for the user.

## M.2 Window Modes

A window can be any actual size on the display. A window can also be off the display entirely, so that it accepts changes to its buffer, but does not display those changes. In addition, a window can be active, inactive, or overlapped, or even completely covered by other windows on the display.

## M.3 Buffered Mode

A window comes up by default in the "buffered" mode. In buffered mode, the program has a "virtual display surface" that it writes to. **windows** maintains this surface in memory, and maintains the actual onscreen window as a scrolled window on that. The program is unconcerned with what part of its screen is being displayed, or even if it is displayed at all. The user operates the window using the frame controls.

Buffered mode is designed to allow the program to be unconcerned with the management of the display. However, the program can set the size of the virtual display using **sizbuf[g](f, x, y)**. When a buffer is resized, its contents is cleared to the current background color.

The buffer is a display surface that the buffer handler keeps for the program. The buffered mode allows a program to "think" it has a window of size N, but the appearance of the window on the actual display is a window on that virtual display that is maintained by the buffer handler.

Because from the programs point of view the window size does not change, and is always the complete window, the resize events are performed transparently to the program. Similarly, the minimize and maximize events are handled for the program (see M.4 “Unbuffered Mode”). Discovery is when a window or part of a window is uncovered on the display.

When in buffered mode, the window manager is may use scroll bars to display within a screen view that is smaller than the buffer.

Although buffered mode automatically handles window status events (covered in M.4 “Unbuffered Mode”), these events are still sent through. For maximum compatibility, the using program should ignore any events not relevant to the mode it is in.

## M.4 Unbuffered Mode

The program can leave buffered mode by using **buffer(f, b)**. Unbuffered mode has no display buffer, and no default actions for events. Instead, the onscreen appearance of the window is set dynamically by the program.

When unbuffered mode is entered, the buffer for the window is discarded, and it becomes entirely up to the program to manage its own window by watching events. The size of the window no longer reflects the buffer, but instead is the size of the onscreen window. In contrast to buffered mode, this can change many times as the window is resized under user control.

Buffered mode can be reentered at any time. The contents of the buffer are cleared to spaces, or “background color”.

When running unbuffered, it is assumed that the program will manage the update of the on-screen display surface itself. When running unbuffered, the program handles events that are normally handled automatically.

The **etredraw** event contains the limits of an update rectangle that shows what part of the client area needs to be redrawn on the screen. The program can ignore the update rectangle, and simply redraw the entire screen, or it may only redraw the figures that overlap the update rectangle. The latter is usually more time efficient. **etredraw** gives its update rectangle in character coordinates. There is also a pixel version of that, **etredrawg**, which uses pixel coordinates. On a graphical system, both messages will be sent, and they duplicate each other. The character coordinate **etredraw** will contain a rectangle of characters that at least covers the pixels that need to be redrawn in a graphical system. Since **etredraw** and **etredrawg** duplicate each other, only one of them should be obeyed, and the other ignored.

The **etresize** event indicates that the onscreen window has changed its size. Its purpose is to let the program update any calculations based on the size of the window. This event can be ignored, used to update stored **maxx[g]** and **maxy[g]** values, or it could even cause the entire arrangement of the screen to be reformed.

**etresize** is not normally used to redraw areas of the screen. If a resize event causes new screen area to be uncovered, then one or more **etredraw** events will also be sent. There is no way to determine exactly which redraw operations correspond to the resize event, so some redundancy is possible with applications that repack their window's contents on a resize.

The **etmin** event indicates that the window has been minimized. When a window is minimized, it will not be displayed, and won't receive **etredraw** events. On the desktop, the window is represented by a small icon. An alternative form of minimization is for the window to receive an **etmin** message, followed by a **etresize** message, and continued **etredraw** messages. This a hint for the window to display vital information in a much smaller format that fits into the minimized icon.

The **etmax** event indicates that the window has been maximized, or covers the entire desktop. If the window needs to be updated, this event may be accompanied by **etresize** and **etredraw** events.

The **etnorm** event indicates a normal window mode has been resumed from **etmin** or **etmax**.

## M.5 Defacto transparency

Using unbuffered mode, it is possible to achieve window transparency. Transparency means that some parts of the window will show through to the display surface underneath the window. This can be used to implement advanced effects like non-rectangular windows, and it is the basis for widgets (which appear in N “Widget Library”). For defacto transparency to work, the drawing order of windows must be tightly controlled.

## M.6 Delayed Window Display

When a window is created, it is not displayed until an actual write operation is made to it. This allows the window to be changed in configuration by the program before it is displayed, and prevents the window being rapidly changed on screen as that happens. For example, the program might want to take the window out of buffered mode, change its size, remove the frame, or add menus.

## M.7 Window Frames

Under most window systems, each window has a "window frame", which has various window management functions. These include:

- Move bars to resize and move the window.
- A title bar that indicates what program is running in the window.
- A "minimize" button.
- A "maximize" button.
- A "menu" bar.
- Vertical and horizontal scroll bars.

When a window is in buffered mode, none of the frame features are under the control of the program, but instead are managed by the buffering software.

The appearance, or even the existence, of any of these items can be customized by the program. The frame can be enabled or disabled by **frame(f, e),** where **f** is the window file, and **e** is true to enable the frame. The title can be set by **title(f, e)**. The title, minimize and maximize buttons are considered part of the "system bar", which can be enabled or disabled by **sysbar(f, e)**. The resize bars are enabled or disabled by **sizable(f, e)**. The scroll bars are enabled and disabled by **scrollv(f, e)** and **scrollh(f, e)**.

The frame control functions **frame**, **title**, **sysbar** and **sizable** are optional within an implementation. The system may not allow an application to, for example, disable its sizing bars, or it may not have a set of system controls on the window. The program must be ready for these commands not to have an effect. For example, after the sizeable function is set false, it should still be ready to receive a new size. If, for example, the size of the graphic to be presented is a fixed size, it would be filled out or clipped to fit the resulting window.

## M.8 Scroll Bars

A window can optionally have vertical and/or horizontal scroll bars. The actual meaning of the scroll bars is up to the program. Each of the scroll bars can generate several events. The vertical scroll bar generates **etsclvul** and **etsclvdl** for line up and line down, and **etsclvup** and **etsclvdp** for page up and page down. The horizontal scroll bar generates **etsclhll** and **etsclhrl** for line left and line right,

and **etsclhlp** and **etsclhrp** for page left and page right. The meaning of these messages are oriented around text. Line movements mean to move a single character (even though for left right the term “line” is close to a misnomer). Page movements mean to move a complete screens worth.

When the slider bar for a scroll bar is moved, the new position is indicated with **etsclvpos** and **etsclhpos**, for vertical and horizontal scroll bar positions. The position is given as a **maxint** ratio, where 0 means the top or left of the scroll bar, and **maxint** means the bottom or right of the scroll bar.

It is common that when one or both of the scroll bars is present on a window that the space where they intersect is occupied by a “move tab”. This serves to resize the window in two directions, and generates standard resize messages to the window.

## M.9 Multiple Windows

If the **output** file appears in the program header, then that is output as the main window by default. A window can also be explicitly opened by **openwin(inw, outw, id)**. When a new window is opened, it is placed on the desktop with the other windows. The text files **inw** and **outw** specify the input and output files attached to the window. The input file receives all user input and events from the window, and all of the write and other drawing operations are sent to the output file. The **id** is a number, from 1 to N, that specifies the logical window. The logical window number 1 is reserved for the standard input and output pair, even if it is not specifically opened, since it can be opened by the program at any time.

The output side of a window must always be unique, but the input side can be shared. Multiple windows can be attached to a single input file, and that input file will receive all of the input and events from all attached windows. The logical window number is returned with each event, so that the source of the input or event can be determined. This forms the basis of “class window handling” (M.13 “Class Window Handling”), or using one input handler as to handle multiple windows.

To completely decouple separate windows, another thread is used to run the separate window. This can be done, for example, to create a widget library that does not depend on the event handler of the calling module.

Windows can be closed using the standard Pascaline **close** call. Only the output side of a window is used to close that window. The input side is automatically closed when there is no longer a window that references it.

## M.10 Parent/Child Windows

Windows systems are tree structured, and each window is a child of a parent window, with the exception of a "root" or master window, which is usually the window that contains the entire desktop. The methods introduced create windows that live side by side on the desktop, and have the desktop as their parents. However, it is possible to nest windows within each other.

A window is opened as a child by **openwin(inw, outw, par, id)**, which is the same **openwin** call with a parent specified. The parent file **par** is the text file that is attached to the output window of the parent. To be a child window means to move as one within it. It maintains its position within the parent, even as the parent itself moves. It always is in front of the Z ordering for the parent. It is minimized, maximized and closed with the parent.

All windows have their position given in relative terms to the parent's client area origin. Any position operation is also relative to the parent.

The common use of the parent is to create "sibling" windows, or windows equivalent to the default program window in status, that are positioned independently within the parent. A program starts as the child of an external parent window. This could be an actual program, such as a program acting as a manager, or it could be the desktop root. There is no difference between these for the program.

Programs do not have direct access to the parent window in which the program was created. That parent window is usually the desktop.

## M.11 Moving and Sizing Windows

Moving a window is done with the **setpos[g](f, x, y)** procedure. The position is given in parent coordinates. When a window is on the desktop, it is necessary to find the size of the desktop, which is found with **scnsiz[g](f, x, y)**. When the desktop size is returned for character sizes, this is calculated using the current character sizes for the window. This is for comparative purposes only. The desktop may use a completely different set of characters and sizes. The purpose of giving the parent sizes is only to allow the program to determine the relative size and placement of the child window in the parent.

Sizing a window is done with **setsiz[g](f, x, y)** procedure. This sets the size of the entire window, and so does not indicate the resulting size of its client area. To find out what window size is needed for a given client area, before actually sizing the window, the function **winclient[g](f, cx, cy, wx, wy, ms)** is used. It returns the window size needed to contain that client. **winclient[g]** takes a set of the current modes of the window to enable it to make the size determination.

The mode set declaration is:

```
{ windows mode sets }
winmod = (wmframe,    { frame on/off }
          wmsize,     { size bars on/off }
          wmsysbar,   { system bar on/off }
          wmscrollv,  { vertical scroll bar on/off }
          wmscrollh); { horizontal scroll bar on/off }
winmodset = set of winmod;
```

To find the current size of a window, in parent terms, the procedure **getsiz[g](f, x, y)** is used. This procedure is useful when you need to find the size of a window on the desktop.

## M.12 Z Ordering

The Z ordering, or back to front ordering of windows, is determined by default to be newest created windows to the front, oldest to the back. This order can be modified by the users when they select windows towards the back to come to the front. The program can also reorder windows at will by sending them to the front or back. Note that the Z ordering is always relative within a parent.

To send a window to the front, the call **front(f)** is used. To send a window to the back, the call **back(f)** is used. To achieve a specific order within a set of windows, they should be send to the front in the opposite order from which they are to appear, i.e., the backmost first, and the frontmost last. Alternately, the windows can be sent to the back in the order they appear.

There are other reasons besides reordering windows that these functions might be useful. When an error in encountered, it is common to send a window containing the error message to the front of the parent Z order, and to send the entire window to the front. Placing an active display or a background pattern in a frameless maximized window can create wallpaper. Placing the same window in the back can be a screen saver.

## M.13 Class Window Handling

**windows** handles windows as a set of two files, the input and output. However, these don't need to always be a set. It is possible to reuse an already open input file as the input side of any new window. This is possible because the user id supplied when the window is opened is passed with all messages to the event procedure.

Allowing a single program thread to handle multiple open windows allows the creation of multiple window programs without the need to create a multitask program. For each message, the id is examined, and the action performed on the appropriate output file.

Because the output file is unique, it is always the handle used to refer to the window in management calls.

## M.14 Parallel Windows

Parallel tasking is a natural match to a multiwindowing system. With a task created to operate each window, the program is simplified, and user response is generally better.

To increase the responsiveness of user interfaces, the recommended program design for windowing programs is to create two tasks for each window, the "foreground" and the "background" tasks. The foreground task performs the event loop and the redraw, resize, move and other user interface tasks. The background task would handle computation tasks and other tasks related to performing actions or changing the drawing surface. The foreground task determines if an action that requires the background task is required, then sends a command to the background task via a monitor queue or other IPC (Interprocess communication).

The reason for this construction is that the things the user sees, such as keeping the client area redrawn, or obeying a move, resize, minimize or similar command always have a task waiting to perform them. This means that elementary user desktop management appears responsive to the user, while data manipulation and computation tasks simply delay client area updates. Users will be much more willing to tolerate client area slowdowns than having a window apparently lock stubbornly in position.. Ideally, the client area should also have an indication that computation is taking place. The most modern method for this is a progress indicator that gives estimates as to when a task will complete, if the task will take longer than about 1 second.

A parallel foreground task is automatically provided for a buffered mode window. It needs to be created in the case of an unbuffered window.

If the work performed in a client area is trivial, only a foreground task need be provided. But be aware that it is very easy to slip into a mode where essential updates are held off. Calling a file procedure, waiting on a timer, or other simple task compromises the response time for window management. Better to leave the window in buffered mode, or structure with foreground/background from program creation than to have to add it later.

## M.15 Menus

The menu is a series of buttons labeled with text for user action. The buttons can either have a direct effect on the program, or they can activate a series of submenus in a tree structure.

The exact location of a menu is system dependent. It may or may not affect the client area.

A menu is described to **windows** by constructing a data structure:

```
menuptr = ^menurec;
menurec = record

              next:    menuptr; { next menu item in list }
              branch:  menuptr; { menu branch }
              onoff:   boolean; { on/off highlight }
              oneof:   boolean; { "one of" highlight }
              bar:     boolean; { place bar under }
              id:      integer; { id of menu item }
              face:    pstring  { text to place in button }

            end;
```

The menu is activated by **menu(f, m)** where **m** is the data structure for the menu.

Menu buttons are highlighted when pointed to. Additionally, a button can have "on/off" highlighting. In this mode, a state is kept for the button that is flipped on or off as the button is pressed. The highlighting for on and off states is different. The data structure that defines a menu is not used or updated after it is used to create a menu.

The **onoff** field true selects on/off highlighting. After the menu is activated, the state of the button can be changed with **menusel(f, id, e)**. The button state is not automatically changed by a user menu select. The program must specifically change the state of the button.

Another highlight mode is "one of" or "checklist" highlighting. In this case, a group of related buttons are joined by setting the **oneof** flag on each item in the list but the last. The highlighting for a button that is selected is different from one that is not, and only one item will be active in the list.

Like on/off buttons, user selection does not automatically change the state of the buttons in the list. It must be specifically changed by a **menusel** call.

Typically, neither on/off nor one/of switching is available for top level (horizontal) menus, and the program should not count on them.

Menu items can be grouped in a vertical list by setting the **bar** field active. This causes a horizontal bar to be drawn under the menu item. Vertical lists are described next in "menu sublisting".

A button can be enabled or disabled. A disabled button is highlighted specially, typically as greyed out. It indicates a button that is entirely inactive, because that function is not available. This is done to allow the same menu to be used regardless of the availability of the functions on the menu. Also, some functions come and go. For example, a "save" button (for "save file") might only be active if the file has changed, and needs to be saved.

The enable status can be changed after the menu is activated by **menusel**.

Each button in a menu can have a numeric **id**. This is used by several functions to change the state of buttons. It is an error to have two buttons with the same id.

The **face** text string indicates what text will appear on the face of the button. The system will automatically size the buttons to fit the largest text of a button, so care should be taken not to have a single button's text so long as to cause a lot of white space in the menu.

### M.16 Setting Menu Active

A menu is constructed by the program as a list using the menu data structure, then set active by calling **menu**, which can also turn the menu back off.

When a menu data structure is activated, all of the fields are copied and placed into an internal form. After the activation, the menu data has no function, and changing its fields will have no effect. The menu data structure can be recycled immediately after the **menu** call.

### M.17 Setting Menu States

For an on/off button, the procedure **menuena(f, id, e)** is used to enable or disable the button after it has been placed in a menu. For **oneof** highlighting, the procedure **menusel** is used. This removes any other select active, and either sets the given button active, or no button active.

### M.18 Standard Menus

Besides presenting a menu in the method standard for the implementation, it is also likely that there exists a standard arrangement of commonly used buttons in the menu. **windows** defines a series of such standard buttons.

```
{ standardized menu entries }

smnew       = 1; { new file }
smopen      = 2; { open file }
smclose     = 3; { close file }
smsave      = 4; { save file }
smsaveas    = 5; { save file as name }
smpageset   = 6; { page setup }
smprint     = 7; { print }
smexit      = 8; { exit program }
smundo      = 9; { undo edit }
smcut       = 10; { cut selection }
smpaste     = 11; { paste selection }
smdelete    = 12; { delete selection }
smfind      = 13; { find text }
smfindnext  = 14; { find next }
smreplace   = 15; { replace text }
smgoto      = 16; { goto line }
smselectall = 17; { select all text }
smnewwindow = 18; { new window }
smtilehoriz = 19; { tile child windows horizontally }
smtilevert  = 20; { tile child windows vertically }
smcascade   = 21; { cascade windows }
smcloseall  = 21; { close all windows }
smhelptopic = 22; { help topics }
smabout     = 23; { about this program }
smmax       = 23; { maximum defined standard menu entries }

{ standard menu selector }

stdmenusel = set of smnew..smmax;
```

Standard menu button definitions should be used whenever possible. To create a menu using standard menu buttons, the procedure **stdmenu** is used. The arrangement of the menu is chosen to match the implementation, and the non-standard buttons provided are integrated into the menu in the normal place for the implementation. When standard buttons are used, the button ids are set to the values shown above, so these values should not be used by the program.

Note that **stdmenu** simply constructs a menu, it does not set it active.

### M.19 Menu Sublisting

When there is not enough room to represent the entire menu on a window, or anytime the size of a menu must be compressed, the tree structure of menus can be used with "pulldown" menus. A pulldown menu is a vertical menu list that appears under the selected button. This is done by placing a submenu in the **branch** pointer of the menu structure. The branch defines the pulldown menu items. The branch menu item is specially indicated to show that it is a branch, and not an immediate button, usually with an arrow pointing towards where the branch pulldown will appear. Any number of levels of pulldown menus can be created. The levels under the topmost branch are generally placed to the right of the branch button.

If a branch pulldown cannot be placed where it should, due to the proximity of the button to the edge of the display, it instead goes back in the other direction, up or down, or both, until space is found for it. If the entire pulldown cannot be displayed, it is truncated. This would only happen if the pulldown was too large for the display.

When a branch button is pressed, no event is generated for it. The button is strictly used to activate the pulldown.

## M.20 Advanced Windowing

The features of a window besides the client area are designed to be the minimum features implemented across platforms. To implement more advanced windows appearances with custom menus, controls and status lines, the standard method is to turn off frames and menus for child windows, and use them as building blocks for more advanced client areas with multiple subwindows and features.

It is recommended that at least one menu appear for a program, even if the functions duplicate some of the advanced controls provided. The reason for this is that the menu has different presentation methods in different systems, so providing the standard "top" menu will help programs' portability.

## M.21 Events

New event types are added for the management mode. As usual, events beyond the definition here should be ignored.

```
module windows;

type

{ events }

evtcod = (etchar,     { ANSI character returned }
          etup,       { cursor up one line }
          etdown,     { down one line }
          etleft,     { left one character }
          etright,    { right one character }
          etleftw,    { left one word }
          etrightw,   { right one word }
          ethome,     { home of document }
          ethomes,    { home of screen }
          ethomel,    { home of line }
          etend,      { end of document }
          etends,     { end of screen }
          etendl,     { end of line }
          etscrl,     { scroll left one character }
          etscrr,     { scroll right one character }
          etscru,     { scroll up one line }
          etscrd,     { scroll down one line }
          etpagd,     { page down }
          etpagu,     { page up }
          ettab,      { tab }
          etenter,    { enter line }
          etinsert,   { insert block }
          etinsertl,  { insert line }
          etinsertt,  { insert toggle }
          etdel,      { delete block }
          etdell,     { delete line }
          etdelcf,    { delete character forward }
          etdelcb,    { delete character backward }
          etcopy,     { copy block }
          etcopyl,    { copy line }
          etcan,      { cancel current operation }
          etstop,     { stop current operation }
          etcont,     { continue current operation }
          etprint,    { print document }
          etprintb,   { print block }
          etprints,   { print screen }
          etfun,      { function key }
          etmenu,     { display menu }
          etmouba,    { mouse button assertion }
          etmoubd,    { mouse button deassertion }
          etmoumov,   { mouse move }
          ettim,      { timer matures }
          etjoyba,    { joystick button assertion }
          etjoybd,    { joystick button deassertion }
          etjoymov,   { joystick move }
          etterm,     { terminate program }
          etmoumovg,  { mouse move graphical }
          etframe,    { frame sync }
```

```
          etresize,   { window was resized }
          etredraw,   { window redraw }
          etmin,      { window was minimized }
          etmax,      { window was maximized }
          etnorm,     { window set normal }
          etmenus,    { menu item selected }
          etsclvul,   { scroll vertical up line }
          etsclvdl,   { scroll vertical down line }
          etsclvup,   { scroll vertical up page }
          etsclvdp,   { scroll vertical down page }
          etsclvpos,  { scroll vertical bar position }
          etsclhll,   { scroll horizontal left line }
          etsclhrl,   { scroll horizontal right line }
          etsclhlp,   { scroll horizontal left page }
          etsclhrp,   { scroll horizontal right page }
          etsclhpos,  { scroll horizontal bar position }

{ event record }

evtrec = record

   winid: ss_filhdl; { identifier of window for event }
   case etype: evtcod of { event type }

      { ANSI character returned }
      etchar:   (char:                char);
      { timer handle that matured }
      ettim:    (timnum:               timhan);
      etmoumov: (mmoun:                mouhan;   { mouse number }
                 moupx, moupy:         integer); { mouse movement }
      etmouba:  (amoun:                mouhan;   { mouse handle }
                 amoubn:               moubut);  { button number }
      etmoubd:  (dmoun:                mouhan;   { mouse handle }
                 dmoubn:               moubut);  { button number }
      etjoyba:  (ajoyn:                joyhan;   { joystick number }
                 ajoybn:               joybut);  { button number }
      etjoybd:  (djoyn:                joyhan;   { joystick number }
                 djoybn:               joybut);  { button number }
      etjoymov: (mjoyn:                joyhan;   { joystick number }
                 joypx, joypy, joypz:  integer); { joystick coords }
      etfun:    (fkey:                 funky);   { function key }
      etmoumovg: (mmoung:              mouhan;   { mouse number }
                 moupxg, moupyg:       integer); { mouse movement }
      etredraw: (rsx, rsy, rex, rey:   integer); { redraw screen }
      etmenus:  (menuid:               integer); { menu item
                                                   selected }

      etup, etdown, etleft, etright, etleftw, etrightw, ethome,
      ethomes, ethomel, etend, etends, etendl, etscrl, etscrr,
      etscru, etscrd, etpagd, etpagu, ettab, etenter, etinsert,
      etinsertl, etinsertt, etdel, etdell, etdelcf, etdelcb, etcopy,
      etcopyl, etcan, etstop, etcont, etprint, etprintb, etprints,
      etmenu, etterm, etframe, etresize, etmin, etmax,
      etnorm: (); { normal events }
```

```
   { end }

end;

begin ! windows
end.
```

## M.22 Event callbacks

As in **terminal**, the events can also be accessed via a series of **virtual** procedures. Only one new event procedure exists in the **graphics** module over the original event procedures in **terminal**:

```
module windows;

virtual procedure etresize; begin end;
virtual procedure etredraw(rsx, rsy, rex, rey: integer); begin end;
virtual procedure etmin; begin end;
virtual procedure etmax; begin end;
virtual procedure etnorm; begin end;
virtual procedure etmenus(menuid: integer); begin end;
virtual procedure etsclvul; begin end;
virtual procedure etsclvdl; begin end;
virtual procedure etsclvup; begin end;
virtual procedure etsclvdp; begin end;
virtual procedure etsclvpos(p: integer); begin end;
virtual procedure etsclhll; begin end;
virtual procedure etsclhrl; begin end;
virtual procedure etsclhlp; begin end;
virtual procedure etsclhrp; begin end;
virtual procedure etsclhpos; begin end;
virtual procedure etsclhpos(p: integer); begin end;

begin ! graphics
end.
```

## M.23 Window Objects

All of the procedures, functions and other declarations in **windows** are also available in a window object. of the form:

```
module windows;

class window;

var input, output: text; ! input and output files
    er:    evtrec; ! event record
    winid:  ss_filhdl; ! window logical identifier


! Executive methods

procedure cursor(x, y: integer); begin end;
function maxx: integer; begin end;
function maxy: integer; begin end;
procedure home; begin end;
procedure del; begin end;
procedure up; begin end;
procedure down; begin end;
procedure left; begin end;
procedure right; begin end;
procedure blink(e: boolean); begin end;
procedure reverse(e: boolean); begin end;
procedure underline(e: boolean); begin end;
procedure superscript(e: boolean); begin end;
procedure subscript(e: boolean); begin end;
procedure italic(e: boolean); begin end;
procedure bold(e: boolean); begin end;
procedure strikeout(e: boolean); begin end;
procedure standout(e: boolean); begin end;
procedure fcolor(c: color); begin end;
procedure bcolor(c: color); begin end;
procedure auto(e: boolean); begin end;
procedure curvis(e: boolean); begin end;
procedure scroll(x, y: integer); begin end;
function curx: integer; begin end;
function cury: integer; begin end;
function curbnd: boolean; begin end;
procedure select(u, d: integer); begin end;
procedure event; begin end;
procedure timer(i: timhan; t: integer; r: boolean); begin end;
procedure killtimer(i: timhan); begin end;
function mouse: mounum; begin end;
function mousebutton(m: mouhan): moubut; begin end;
function joystick: joynum; begin end;
function joybutton(j: joyhan): joybtn; begin end;
function joyaxis(j: joyhan): joyaxn; begin end;
procedure settab(t: integer); begin end;
procedure restab(t: integer); begin end;
procedure clrtab; begin end;
```

```
function funkey: funky; begin end;
function frametimer(e: boolean); begin end;
procedure autohold(e: boolean); begin end;
procedure wrtstr(s: string); begin end;
function maxxg: integer; begin end;
function maxyg: integer; begin end;
function curxg: integer; begin end;
function curyg: integer; begin end;
procedure line(x1, y1, x2, y2: integer); begin end;
overload procedure line(var f: text; x2, y2: integer); begin end;
overload procedure line(x2, y2: integer); begin end;
overload procedure linewidth(w: integer); begin end;
procedure rect(x1, y1, x2, y2: integer); begin end;
procedure frect(x1, y1, x2, y2: integer); begin end;
procedure rrect(x1, y1, x2, y2, xs, ys: integer); begin end;
procedure frrect(x1, y1, x2, y2, xs, ys: integer); begin end;
procedure ellipse(x1, y1, x2, y2: integer); begin end;
procedure fellipse(x1, y1, x2, y2: integer); begin end;
procedure arc(x1, y1, x2, y2, sa, ea: integer); begin end;
procedure farc(x1, y1, x2, y2, sa, ea: integer); begin end;
procedure fchord(x1, y1, x2, y2, sa, ea: integer); begin end;
procedure ftriangle(x1, y1, x2, y2, x3, y3: integer); begin end;
overload procedure ftriangle(var f: text; x2, y2, x3, y3: integer);
begin end;
overload procedure ftriangle(x2, y2, x3, y3: integer); begin end;
overload procedure ftriangle(var f: text; x3, y3: integer); begin
end;
overload procedure ftriangle(x3, y3: integer); begin end;
procedure cursorg(x, y: integer); begin end;
function baseline: integer; begin end;
procedure setpixel(x, y: integer); begin end;
procedure fover; begin end;
procedure bover; begin end;
procedure finvis; begin end;
procedure binvis; begin end;
procedure fxor; begin end;
procedure bxor; begin end;
function chrsizx: integer; begin end;
function chrsizy: integer; begin end;
function fonts: integer; begin end;
procedure font(fc: integer); begin end;
procedure fontnam(fc: integer; var fns: string); begin end;
procedure fontsiz(s: integer); begin end;
procedure chrspcy(s: integer); begin end;
procedure chrspcx(s: integer); begin end;
function dpmx: integer; begin end;
function dpmy: integer; begin end;
function strsiz(view s: string): integer; begin end;
function strsizp(view s: string): integer; begin end;
```

```
function chrpos(view s: string; p: integer): integer; begin end;
procedure writejust(view s: string; n: integer); begin end;
function justpos(view s: string; p, n: integer): integer; begin end;
procedure condensed(e: boolean); begin end;
procedure extended(e: boolean); begin end;
procedure xlight(e: boolean); begin end;
procedure light(e: boolean); begin end;
procedure xbold(e: boolean); begin end;
procedure hollow(e: boolean); begin end;
procedure raised(e: boolean); begin end;
procedure settabg(t: integer); begin end;
procedure restabg(t: integer); begin end;
procedure fcolorg(var f: text; r, g, b: integer); begin end;
procedure fcolor(var f: text; r, g, b: integer); begin end;
overload procedure fcolor(r, g, b: integer); begin end;
procedure bcolorg(r, g, b: integer); begin end;
procedure bcolor(var f: text; r, g, b: integer); begin end;
overload procedure bcolor(r, g, b: integer); begin end;
procedure loadpict(p: integer; view fn: string); begin end;
function pictsizx(p: integer): integer; begin end;
function pictsizy(p: integer): integer; begin end;
procedure picture(p: integer; x1, y1, x2, y2: integer); begin end;
procedure delpict(p: integer); begin end;
procedure scrollg(x, y: integer); begin end;
procedure title(var f: text; view ts: string); begin end;
overload procedure title(view ts: string); begin end;
procedure buffer(var f: text; e: boolean); begin end;
overload procedure buffer(e: boolean); begin end;
procedure sizbuf(var f: text; x, y: integer); begin end;
overload procedure sizbuf(x, y: integer); begin end;
procedure sizbufg(var f: text; x, y: integer); begin end;
overload procedure sizbufg(x, y: integer); begin end;
procedure getsiz(var f: text; var x, y: integer); begin end;
overload procedure getsiz(var x, y: integer); begin end;
procedure getsizg(var f: text; var x, y: integer); begin end;
overload procedure getsizg(var x, y: integer); begin end;
procedure setsiz(var f: text; x, y: integer); begin end;
overload procedure setsiz(x, y: integer); begin end;
procedure setsizg(var f: text; x, y: integer); begin end;
overload procedure setsizg(x, y: integer); begin end;
procedure setpos(var f: text; x, y: integer); begin end;
overload procedure setpos(x, y: integer); begin end;
procedure setposg(var f: text; x, y: integer); begin end;
overload procedure setposg(x, y: integer); begin end;
procedure scnsiz(var f: text; var x, y: integer); begin end;
overload procedure scnsiz(var x, y: integer); begin end;
procedure scnsizg(var f: text; var x, y: integer); begin end;
overload procedure scnsizg(var x, y: integer); begin end;
```

```
procedure winclient(var f: text; cx, cy: integer; var wx, wy:
integer; view ms: winmodset); begin end;
overload procedure winclient(cx, cy: integer; var wx, wy: integer;
view ms: winmodset); begin end;
procedure winclientg(var f: text; cx, cy: integer; var wx, wy:
integer; view ms: winmodset); begin end;
overload procedure winclientg(cx, cy: integer; var wx, wy: integer;
view ms: winmodset); begin end;
procedure front(var f: text); begin end;
overload procedure front; begin end;
procedure back(var f: text); begin end;
overload procedure back; begin end;
procedure frame(var f: text; e: boolean); begin end;
overload procedure frame(e: boolean); begin end;
procedure sizable(var f: text; e: boolean); begin end;
overload procedure sizable(e: boolean); begin end;
procedure sysbar(var f: text; e: boolean); begin end;
overload procedure sysbar(e: boolean); begin end;
procedure scrollv(var f: text; e: boolean);
overload procedure scrollv(e: boolean);
procedure scrollh(var f: text; e: boolean);
overload procedure scrollh(e: boolean);
procedure scrollposv(var f: text; p: integer);
overload procedure scrollposv(p: integer);
procedure scrollposh(var f: text; p: integer);
overload procedure scrollposh(p: integer);
procedure scrollsizv([var f: text;] s: integer);
overload procedure scrollsizv(s: integer);
procedure scrollsizh([var f: text;] s: integer);
overload procedure scrollsizh(s: integer);
procedure menu(var f: text; m: menuptr); begin end;
overload procedure menu(m: menuptr); begin end;
procedure menuena(var f: text; id: integer; onoff: boolean); begin
end;
overload procedure menuena(id: integer; onoff: boolean); begin end;
procedure menusel(var f: text; id: integer; select: boolean); begin
end;
overload procedure menusel(id: integer; select: boolean); begin end;
procedure stdmenu(view sms: stdmenusel; var sm: menuptr; pm:
menuptr); begin end;

! Event callbacks

virtual procedure evchar(c: char); begin end;
virtual procedure evtim(t: timhan); begin end;
virtual procedure evmoumov(m: mouhan); begin end;
virtual procedure evmouba(h: mouhan; b: moubut); begin end;
virtual procedure evmoubd(h: mouhan; b: moubut); begin end;
virtual procedure evjoyba(h: joyhan; b: joybut); begin end;
```

```
virtual procedure evjoybd(h: joyhan; b: joybut); begin end;
virtual procedure evjoymov(h: joyhan; x, y, z: integer); begin end;
virtual procedure evfun(k: funky); begin end;
virtual procedure evup; begin end;
virtual procedure evdown; begin end;
virtual procedure evleft; begin end;
virtual procedure evright; begin end;
virtual procedure evleftw; begin end;
virtual procedure evrightw; begin end;
virtual procedure evhome; begin end;
virtual procedure evhomes; begin end;
virtual procedure evhomel; begin end;
virtual procedure evend; begin end;
virtual procedure evends; begin end;
virtual procedure evendl; begin end;
virtual procedure evscrl; begin end;
virtual procedure evscrr; begin end;
virtual procedure evscru; begin end;
virtual procedure evscrd; begin end;
virtual procedure evpagd; begin end;
virtual procedure evpagu; begin end;
virtual procedure evtab; begin end;
virtual procedure eventer; begin end;
virtual procedure evinsert; begin end;
virtual procedure evinsertl; begin end;
virtual procedure evinsertt; begin end;
virtual procedure evdel; begin end;
virtual procedure evdell; begin end;
virtual procedure evdelcf; begin end;
virtual procedure evdelcb; begin end;
virtual procedure evcopy; begin end;
virtual procedure evcopyl; begin end;
virtual procedure evcan; begin end;
virtual procedure evstop; begin end;
virtual procedure evcont; begin end;
virtual procedure evprint; begin end;
virtual procedure evprintb; begin end;
virtual procedure evprints; begin end;
virtual procedure evmenu; begin end;
virtual procedure evterm; begin end;
virtual procedure evmoumovg(m: mouhan); begin end;
virtual procedure evframe; begin end;
virtual procedure evresize; begin end;
virtual procedure evredraw(rsx, rsy, rex, rey: integer); begin end;
virtual procedure evmin; begin end;
virtual procedure evmax; begin end;
virtual procedure evnorm; begin end;
virtual procedure evmenus(menuid: integer); begin end;
virtual procedure evsclvul; begin end;
```

```
virtual procedure evsclvdl; begin end;
virtual procedure evsclvup; begin end;
virtual procedure evsclvdp; begin end;
virtual procedure evsclvpos(p: integer); begin end;
virtual procedure evsclhll; begin end;
virtual procedure evsclhrl; begin end;
virtual procedure evsclhlp; begin end;
virtual procedure evsclhrp; begin end;
virtual procedure evsclhpos(p: integer); begin end;

begin ! window

end.

class childwindow(var parent: text);

extends window;

begin ! childwindow

end.

begin ! windows

end.
```

A **window** object can be created as follows:

```
program p;

joins windows;

var wi: windows.window;

begin

   wi.home; { send cursor to home position }
   writeln(wi.output, 'hello, windows world');
   repeat { event loop }

      wi.event; { get next event }
      { process events }

   until wi.er.type = etterm { loop until program cancelled }

end.
```

Where **wi** is the **window** object. **window** objects can be instantiated statically or dynamically.

**childwindow** objects can be created using several methods:

```
program p;

joins windows;

var mywindow(output): terminal.childwindow;

begin
end.
```

Creates a child window of the program main window attached to the program parameter **input** and **output** files.

```
program p;

joins windows;

var inwin, outwin: text;

procedure doit;

var mywindow(outwin): terminal.window;

begin

end;

begin

   openwin(inwin, outwin, 1);
   doit

end.
```

Creates a child window of the procedurally created window attached to the pair **inwin** and **outwin**. Note that the child window is not instantiated until the parent window is created.

```
program p;

joins windows;

var mywindow: windows.window;
    mychildwindow(mywindow.output): windows.childwindow;

begin

end.
```

Creates a subwindow of a **window** object.

**Window** objects contain their own state for the following:

1. Location of cursor.
2. Current attributes and colors.
3. Current font and size.
4. Buffer working/display status.
5. Tab stops.
6. Timer numbers.
7. Cached pictures.
8. Buffered status.
9. Sizes and positions.
10. Title.
11. Modes.
12. Z ordering.
13. Menus.

When a **window** object is instantiated, it creates a new window parented by the root or desktop. It contains an input file, an output file, and an event record, all publically accessible. The **input** and **output** files can be used to operate on the created window using the procedural interface. Because a window object has its own **input** and **output** files, it will not receive any window events outside of the ones relevant for the contained window. In addition, the **window** object contains its logical identifier as **winid**, which can be used to refer to the window logical identifier procedurally.

An extended class, **childwindow**, exists with all the members of window, but takes a parent file as a constructor parameter. This class creates window objects that are a child of the specified parent window.

Each **window** object gets the complete state of the module **windows** when it is created.

The complete catalog of procedures and functions from **windows** are available as methods in the window class. There are minor differences, which are detailed with the procedure and function descriptions in M.25 “Procedures and Functions in **windows**”.

The input and output files used by the **window** object are made available as class members input and output. This allows them to be used in non-method procedures (such as read and write).

A **window** object also contains an event record, so it is not necessary to specify an external event record in the **event** procedure.

Window objects are inherently separate drawing surfaces (unlike **screen** or **surface** objects).

## M.24 Exceptions

The following exceptions are generated in **windows**:

| Identifier | Meaning |
|---|---|
| **TooManyFiles** | The total number of open files possible was exceeded. |
| **NoJoyStick** | No joystick access was available. |
| **NoTimer** | No timer access was available. |
| **ToManyTimers** | The total number of timers available was exceeded. |
| **CannotPerformSpecial** | Cannot perform operation on special file. A system file cannot be positioned, etc. |
| **FilenameEmpty** | Filename specified was empty. |
| **InvalidScreenNumber** | Screen number specified was invalid. |
| **InvalidHandle** | An invalid handle was specified. |
| **InvalidTab** | An invalid tab position was specified. |
| **CannotCreateScreenBuffer** | Cannot create a buffer for the screen. |
| **CannotQueryJoystick** | Could not get information on joystick. |
| **InvalidJoystickHandle** | Invalid handle specified for joystick. |
| **InvalidTimerHandle** | Invalid handle specified for timer. |
| **CannotWriteDirect** | Cannot write direct string with **auto** on. |
| **CannotPositionTextByPixel** | Cannot position text by pixel with **auto** on. |
| **CannotPositionOutsideScreen** | Cannot position outside screen with **auto** on. |
| **CannotReenableAutoOffGrid** | Cannot re-enable **auto** off grid |
| **CannotReenableAutoOutsideScreen** | Cannot re-enable **auto** outside screen. |
| **InvalidFontNumber** | Invalid font number. |
| **NoValidTerminalFont** | Valid terminal font not found. |
| **CannotResizeFontWithAuto** | Cannot resize a font with **auto** enabled. |
| **CannotChangeFontsWithAuto** | Cannot change fonts with **auto** enabled. |
| **InvalidFontNumber** | Invalid font number. |
| **NoFontForNumber** | Logical font number has no font assigned. |
| **CannotSizeTerminalFont** | Cannot change the size of the terminal font. |
| **TooManyTabs** | Too many tabs are set. |
| **CannotTabGraphicalWithAuto** | Cannot set a graphical tab with **auto** enabled. |
| **StringIndexOutOfRange** | String index out of range. |
| **PictureFileNotFound** | No picture file was found by filename. |
| **PictureFilenameToLarge** | The specified picture filename was too large. |
| **CannotJustifySystemFont** | Cannot justify system font. |
| **FileNotAttachedToWindow** | Text file not attached to a window. |
| **WinIdInUse** | Windows logical identifier in use. |
| **FileInUse** | Text file is already in use. |
| **InputFileMode** | Input file in wrong mode. |
| **InvalidSize** | Invalid screen buffer size specified. |
| **WindowNotBuffered** | Window is not in buffered mode. |
| **InvalidWindowNumber** | Invalid window logical identifier number. |
| **InvalidScrollBarPosition** | Invalid scroll bar slider position. |
| **InvalidScrollBarSize** | Invalid scroll bar size. |

**Windows** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **windows** will go back to **windows**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **windows** need only capture the exceptions occurring in the procedure or function that is called.

## M.25 Procedures and Functions in windows

For all of the following module calls, If the screen file **f** is not present, the default is the standard output file. If the procedure or function is a method, the output screen file should not be specified, since it is inherent in the object.

procedure openwin(infile, outfile [, parent]: text; id: integer);

Opens a new window. The input file **infile** will get messages pertaining to the window, and the output file **outfile** will be used to draw to it. The input file may already be open elsewhere, in which case it will get all messages for all open windows opened with it. The window will be placed at 1,1 within the parent and held out of display. The output file is always used as the window handle, since it unique to the window.

The window identifier **id** is a number from 1 to n that is returned in messages concerning the window.

Only the output side of a window pair is used in procedures or functions that operate on the window once opened with **openwin**.

A window is closed with the standard Pascaline close procedure, used on the output file for the window. The input side of the window is automatically closed when there are no longer any windows that reference it.

If the optional parent window is specified, the new window is created as a child of the given parent. This means that it will always be displayed within the parent, and be clipped to it. If the parent is not specified, it defaults to the **output** file.

Exceptions: **InvalidWindowNumber, WinIdInUse, FileInUse, InputFileMode**

procedure buffer([var f: text;] b: boolean);

Engages or removes window **f** from buffered mode, according to the boolean **b**. In buffered mode, all of the drawing for a window is performed on a memory buffer, then copied to the screen. The screen view of the buffer can be all or part of the buffer, and multiple buffers can be managed.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure sizbuf[g]([var f: text;] x, y: integer);

Sets the size of the buffer used to draw into. **x** and **y** indicate the width and height, respectively, of the buffer surface in window **f**. It is an error if buffering is not enabled. **sizbuf** sets the buffer size in characters. **sizbufg** sets the buffer size in pixels.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure title([var f: text;] view s: string): integer;

Sets the title of the window **f** to the string **s**. If the title is too long for the current window size, an implementation defined method will be used to make it fit, for example, it is clipped.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure frame([var f: text;] e: boolean);

Enables or disables the appearance of the frame in window **f**, according to boolean **e**, which includes the minimize, maximize, title, size, move and close controls. If the frame is removed, the user will be unable to operate the frame controls.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure sysbar([var f: text;] e: boolean);

Enables or disables the system control bar for a window **f**. If **e** is true, the system bar is enabled, otherwise disabled. The system bar normally includes the title, minimum, maximum, and other control buttons. If the frame is not enabled, then the system bar will not appear regardless of the **sysbar** status, but it will be recorded.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure sizable([var f: text;] e: boolean);

Enables or disables the sizing bars for a window **f**. If **e** is true, the sizing bar is enabled, otherwise disabled. If the frame is not enabled, then the size bars will not appear regardless of the sizeable status, but it will be recorded. If the size bars are removed, the user will be unable to resize the window.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure scrollv([var f: text;] e: boolean);

Enables or disables the vertical scroll bar for a window **f**. If **e** is true, the vertical scroll bar is enabled, otherwise disabled.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure scrollh([var f: text;] e: boolean);

Enables or disables the horizontal scroll bar for a window **f**. If **e** is true, the horizontal scroll bar is enabled, otherwise disabled.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure scrollposv([var f: text;] p: integer);
procedure scrollposh([var f: text;] p: integer);

Sets either the vertical or horizontal scrollbar slider position for window **f** to position **p**. The position is in ratioed **maxint** format. That is, 0 means to set the position to the top or left, and **maxint** means bottom or right. The position is affected by the size of the scrollbar slider. For example, if the slider occupies %50 of the scrollbar, then the range of positions would only be from 0 to **maxint** div 2. If the position given is beyond the maximum position possible, then the slider is set to the maximum travel position, and no error occurs. It is an error if the position is negative.

The program must specifically set the position of the scrollbar. The user moving the scrollbar slider may temporarily move the slider while it is being moved, but this will not remain in position after the user releases it. The program must specifically set the scrollbar position in response to the event.

If the window file **f** does not exist, the **output** file will be used by default.

Exceptions: **InvalidHandle, FileNotAttachedToWindow, InvalidScrollBarPosition**

procedure scrollsizv([var f: text;] s: integer);
procedure scrollsizh([var f: text;] s: integer);

Sets either the vertical or horizontal scrollbar slider in window **f**, to the size **s**. The size of the scrollbar slider is a **maxint** ratio, with 0 meaning infinitely small, and **maxint** meaning that it occupies the entire scrollbar. In practice, there is a practical limit to how small the slider can be. If the slider is set too small, it will be set to the minimum size, and no error will occur. If the size set is negative, then an error will result.

The size of the scrollbar is set to a reasonable default if it is never specifically set. This is typically a fairly small size that is still easy to press and manipulate by the user. This allows the scrollbar to be used when slider sizing is not supported by the program.

The meaning of the scrollbar slider size is up to the program. However, it is typically used to indicate how much of the data is onscreen. For example, if a document has %50 of its content currently displayed, then the slider would be set to %50 of the scrollbar.

If the window file **f** does not exist, the "output" file will be used by default.

Exceptions: **InvalidHandle, FileNotAttachedToWindow, InvalidScrollBarSize**

procedure setpos[g]([var f: text;] x, y: integer);

Sets the position, within the parent, of a child window **f**, using position **x** and **y**. If the window is on the desktop, then the window position is relative to the desktop. The **setpos** procedure sets the position in terms of characters, and the **setposg** procedure set the position in terms of pixels.

The position is set in terms of the parent's coordinates. The mode of the parent's coordinates may not be known. For example, the desktop may not even have a character mode, so the idea of a character size may be arbitrary. What matters in this case is the relative position and size in the desktop as determined by **scnsiz**.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure scnsiz[g]([var f: text;] var x, y: integer);

Finds the size of the user screen or desktop for window **f** to the size **x** and **y**. **scnsiz** returns the size in character terms, and **scnsizg** returns the size in pixel terms. If the desktop does not have a character mode, an arbitrary scale is created. What is important is the relative location within the desktop.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure winclient[g]([var f: text;] cx, cy: integer; var wx, wy: integer; ms: modset);

Determines the window size needed for a given client size within window **f**. Given a desired client size of **cx** and **cy**, in width and height, the necessary window size to achieve that will be returned in **wx** and **xy**. **winclient** determines these measurements in character dimensions, and **winclientg** determines them in pixel terms.

The set of modes **ms** is used to determine the needed window size.

If the parent of the window has no character mode, then one is created that will be acceptable to **setpos**.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure getsiz[g](var f: text; var x, y: integer);

Finds the size of a window in parent coordinate terms for window **f**, to size **x** and **y**. **getsiz** returns the character size, and **getsizg** returns the pixel size.

If the parent has no character mode, then one is created that is compatible with other **windows** functions and procedures.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure setsiz[g]([var f: text;] var x, y: integer);

Sets the size of window **f** in parent coordinate terms, to size **x** and **y**. **setsiz** sets the character size, and **setsizg** sets the pixel size.

If the parent has no character mode, then one is created that is compatible with other **windows** functions and procedures.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure back[(var f: text)];

Sends the window **f** to the back of the parent Z order.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure front[(var f: text)];

Sends the window **f** to the front of the parent Z order.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure menu([var f: text;] m: menptr);

Sets up the menu bar for window **f** from the given list, which contains a menu bar definition structure.

If the menu pointer is nil, then the menu is removed.

The menu data structure is copied during the call, so the menu structure can be reused or freed.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure stdmenu(sms: stdmenusel; var sm: menuptr; pm: menuptr);

Constructs a standard menu and returns that in **sm**. **sms** contains the set of desired standard buttons. **pm** contains a menu containing non-standard buttons to be added to the menu. The menu is constructed using the desired standard buttons, and the non-standard buttons placed into the menu at a standard location.

Exceptions: None

procedure menuena([var f: text;] id: integer; e: boolean);

Enables or disables an on/off menu button for window **f**. **id** refers to a button **id** that was specified in the menu data structure. If **e** is true, the button is enabled, otherwise disabled. The highlighting of the button will change to match.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

procedure menusel([var f: text;] id: integer; e: boolean);

Selects a button from a **oneof** list to be active in window **f**. **id** refers to a button id that was specified in the menu data structure. If **e** is true, the button is selected, otherwise deselected. All other buttons in the **oneof** list are deactivated. The highlighting of the buttons will change to match.

Exceptions: **InvalidHandle, FileNotAttachedToWindow**

## M.26 Events and Callbacks In windows

For each item, both the event record section and the virtual procedure is presented. See the description of the event record (L.21 “Declarations”) for the format of the entire record.

Event: etresize
**virtual procedure** evresize;

The window was resized. The new size can be found with **getsiz(f, x, y)**.

Event: etredraw
**virtual procedure** evredraw(rsx, rsy, rex, rey: integer);

Signifies that the rectangle represented by the starting point in the upper left hand corner (**rsx**, **rsy**) and the ending point in the lower right hand corner (**rex**, **rey**) should be redrawn. This redraw can be satisfied by either redrawing just the rectangle, or by redrawing the entire window client area.

It is possible that a complex area needing to be redrawn could be sent as a series of redraw commands.

Event: etmin
**virtual procedure** evmin;

Signifies that the window was minimized. This may not need any action, but is simply for information.

Event: etmax
**virtual procedure** evmax;

Signifies that the window was maximized. This may not need any action, but is simply for information. If the window needs to be redrawn as a result of the change, a separate redraw event will be sent.

Event: etnorm
**virtual procedure** evnorm;

The window was normalized back to its original size. This may not need any action, but is simply for information. If the window needs to be redrawn as a result of the change, a separate redraw event will be sent.

Event: etmenus
**virtual procedure** evmenus(menuid: integer);

A menu item was selected. The **id** parameter gives which menu item was selected, which is a logical identifier number selected when the menu structure was constructed.

Event: etsclvul
**virtual procedure** evsclvul;

The scroll up line button was pressed on the vertical scroll bar. The meaning of line is that one single character is to be moved, as in one line. The meaning can signify one pixel, or one increment of another type.

Event: etsclvdl
**virtual procedure** evsclvdl;

The scroll down line button was pressed on the vertical scroll bar. The meaning of line is that one single character is to be moved, as in one line. The meaning can signify one pixel, or one increment of another type.

Event:
**virtual procedure** evsclvup;

The scroll up page button was pressed on the vertical scroll bar. The meaning of line is that one whole displayed page is to be moved. This means that the page below the current one is to be displayed, if it exists.

Event:
**virtual procedure** evsclvdp;

The scroll down page button was pressed on the vertical scroll bar. The meaning of line is that one whole displayed page is to be moved. This means that the page above the current one is to be displayed, if it exists.

Event:
**virtual procedure** evsclvpos(p: integer);

The scroll bar position was changed. The position p is a rationed **maxint** position, with 0 being the position at extreme top, and **maxint** being the position at extreme bottom. Although the scroll bar slider is allowed to move so that the user will see it reposition, the actual position does not take effect until a **scrollvpos(f, p)** call is made.

Event:
**virtual procedure** evsclhll;

The scroll left line button was pressed on the horizontal scroll bar. The meaning of line is that one single character is to be moved, as in one line, which in the case of the horizontal scroll bar means a column of characters. The meaning can signify one pixel, or one increment of another type.

Event:
**virtual procedure** evsclhrl;

The scroll right line button was pressed on the horizontal scroll bar. The meaning of line is that one single character is to be moved, as in one line, which in the case of the horizontal scroll bar means a column of characters. The meaning can signify one pixel, or one increment of another type.

Event:
**virtual procedure** evsclhlp;

The scroll left page button was pressed on the vertical scroll bar. The meaning of line is that one whole displayed page is to be moved. This means that the page to the right of the current one is to be displayed, if it exists. Note that “page” in this context refers to a displayed area in the client window.

Event:
**virtual procedure** evsclhrp;

The scroll right page button was pressed on the vertical scroll bar. The meaning of line is that one whole displayed page is to be moved. This means that the page to the left of the current one is to be displayed, if it exists. Note that “page” in this context refers to a displayed area in the client window.

Event:
**virtual procedure** evsclhpos(p: integer);

The scroll bar position was changed. The position p is a rationed **maxint** position, with 0 being the position at extreme top, and **maxint** being the position at extreme bottom. Although the scroll bar slider is allowed to move so that the user will see it reposition, the actual position does not take effect until a **scrollvpos(f, p)** call is made.

# N Annex: Widget Library

**terminal** and **graphics** define terminal and graphical operations on a fixed screen. **windows** defines its division into "virtual windows". **Widgets** provides elements placed within those windows to allow user control. These include buttons, sliders, scroll bars, checkboxes, and similar user interface elements.

These are sometimes referred to as controls or widgets. Dialogs are predefined windows with widgets in them. What these user elements have in common is they all use **windows** elements to define the window they appear in, and **graphics** or **terminal** routines to draw their appearance.

**widgets** can be performed entirely in terms of **windows** with **graphics** or **terminal** calls. However, it is still dependent on a particular operating system because of its appearance. **widgets** maintains the "look and feel" of a particular operating system.

## N.1 Tiles, Layers and Looks

Each widget is implemented in its own window. A layout of widgets occurs when several widgets are tiled side by side, or placed on top of each other (layered). For example, a window surface with text and scrollbars to control the positioning of that text is done by constructing a series of windows. The first window is the window that holds the presented text. The second and third are the windows that hold horizontal and vertical scrollbars.

Layering is done by defacto transparency. A series of widgets are placed one atop the other. For example, a text window can be laid on top of a background window.

The key to interface design is to think of a window as simply a building block for construction of the user interface.

Widgets are fundamental to the "look" of an interface. Because **widgets** uses the native widgets on the operating system it serves, the client program will pick up quite a bit of that look from just the use of the widgets.

There is more to an application than just the look of the widgets. There are layout conventions, actions, and other intangibles. The rule that applies to Pascaline portability is:

- Pascaline programs will be able to target a high percentage of simple applications just by use of its normal **widgets** components.
- Pascaline should be able to finish a high percentage of the work to create complex applications.

The typical cycle in designing an Pascaline application is to design an initial version that uses just **widgets** components, then finish the design with special layouts, actions and colors for a particular operating system that give it the "look and feel" of a native application.

## N.2 Background colors and placement

Widgets are placed by specifying their "bounding box" or rectangle. This is the rectangle that contains the widget. A widget can occupy all of the specified bounding box, or just a part of it. Typically, the operating system will try its best to format the widget to fit within the space provided.

The color scheme for widgets can vary. However, widgets are designed to be placed against a background color that is system dependent. This is available as a new system defined color, **backcolor**. It can be selected by the standard color selection routines.

```
type color = (black, white, red, green, blue, cyan,
              yellow, magenta, backcolor);

fcolor(backcolor);
bcolor(backcolor);
```

There are many ways to group widgets so that they blend into your background. The surface they appear on can be specifically colored using the background color, or you can create a child window with that color. Also, there is a background widget that can color the background for widgets automatically.

## N.3 Sizes

Using widgets can go a long way towards getting the look and feel of a system in an independent way. The other important factor for achieving system independence is to control sizing of widgets. For each widget, with the exception of dialogs, there is a size routine. The size routine takes the particular features of that widget, such as face text or borders, and gives the best size for the widget, given the desired client area, contents, etc..

The size information must be considered against the particular widget to be created. Sizing is key to establishing the layout of widgets in a window, and key to producing a truly portable application.

## N.4 Logical Widget Identifiers

A widget is created with a logical number, from 1 to n, where n is a positive integer. You specify the number you wish to create the widget under. The id of a widget cannot be the same as any other active widget, but widget logical identifiers can be reused by killing the widget first.

## N.5 Killing, Selecting, Enabling and Getting Text to and from Widgets

A widget is killed by **killwidget(f, id)**.

Some widgets can be selected, which changes their appearance to the select state. A widget is selected by **selectwidget(f, id)**.

A selected widget can be a checkbox that is checked, a radio button that is pushed, or similar effect. The processing of selects, and keeping track of the state of widgets is up to you. However, this is a very flexible system. You can implement widgets that flip their state when clicked on, a series of widgets that are mutually exclusive, and many other combinations.

Similar to selection, widgets can be enabled or disabled by **enablewidget(f, id).** Widgets are disabled by default. When a widget is disabled, it has the disabled appearance, such as greyed out. The

widget will not give click events when it is disabled. Disabling a widget is typically used when it is not needed or not available in the current context. For example, a "next" button could be disabled when there is no next item to process.

Some widgets have text that can be set, read, or in some cases, both. The text in a widget is set by **putwidgettext(f, id)**. The text in a widget can be read by **getwidgettext(f, id)**.

Setting and getting the widget text is used with widgets that allow the user to modify the text, such as an edit box.

## N.6 Resizing and repositioning a widget

To prevent the need to remove and replace widgets each time a window is resized, **sizwidget[g](f, id, x, y)**. To reposition the widget in the parent window, **poswidget[g](f, id, x, y)** exists.

## N.7 Types of widgets

Widgets come in three different types, controls, components and dialogs. Controls are widgets that the user can manipulate, and these widgets issue events to the program that owns them.

Components are display widgets whose only job is to form part of a display to the user. A group box, and a background are examples of components. Some components have active displays, such as the progress bar. However, components never issue events.

Dialogs are fully autonomous windows that exist apart from the applications windows. They can be very complex inside, having a whole system of layout, widgets and other features. They resemble entirely separate programs.

Dialogs take a series of parameters when they are called, and deliver those same parameters back to the caller, with any modifications the user performs on the data.

## N.8 Z ordering

**widgets** uses “implicit Z ordering” for layered widgets. In order for widgets to properly layer, the drawing Z order must be controlled. For example, a background must be drawn first, followed by whatever is on top of it, or the drawing of the background will wipe out what is on the surface of the background.

To enable this, **widgets** makes sure that widgets which are designed to be layered appear first in the drawing order. Typically, this means that controls are in front of components.

## N.9 Controls

A button can be created with **button[g](f, x1, xy1, x2, y2, s, id)**. The button is drawn in window f, in the specified rectangle (**x1**, **y1**) to (**x2**, **y2**) with a label text string **s**, and logical widget number **id**. The text string will be a single line of text with no control characters, and will be presented on the face of the button. The font style and size will be the same as other buttons in the operating systems user interface.

When a button is pressed, it will typically change its appearance to indicate that. The button will send an event **etbutton**. This event carries the id of the button that was asserted. Similarly, when a button is released, it changes appearance back from the pressed state.

Buttons can be any size, any height and width. The button face text does not get larger with the button, but remains centered. However, if the button too small, some of the face text will be clipped off. The size of the button can be determined before creating it by **buttonsiz[g](f, s, w, h)**.

Buttons cannot be selected, but they can be disabled. The text in a button can be neither read nor written.

A checkbox is created with **checkbox[g](f, x1, y1, x2, y2, s, id)**. when hit, give a single event that indicates activation, **etchkbox**. The event contains the identifier of the widget.

Checkbox sizing is found with **checkboxsiz[g](f, s, w, h)**. Checkboxes are sized to minimum, but since they have no edges (like a button), there is typically no need to add space to them.

Checkboxes can be selected (checked). They can be enabled or disabled, and default to enabled. They cannot have their face text changed or read.

Radio buttons work identically to checkboxes, but have a different appearance. A radio button is created by **radiobutton[g](f, x1, y1, x2, y2, s, id).** Radio buttons, when hit, give a single event that indicates activation, **etradbut**. This event contains the id of the widget.

Radio button sizing is found with **radiobuttonsizsiz[g](f, s, w, h)**. Radio buttons are sized to minimum, but since they have no edges (like a button), there is typically no need to add space to them.

Radio buttons can be selected (checked). They can be enabled or disabled, and default to enabled. They cannot have their face text changed or read.

Scrollbar widgets are identical to the ones used to generate scrolling for windows. However, they are free floating, and can appear anywhere in the window, not just the sides. In addition, the height and width of them can be controlled, instead of being fixed to the window size.

Scrollbars are placed vertically by **scrollvert[g](f, x1, y1, x2, y2, id)**. Scrollbars are placed horizontally by **scrollhoriz[g](f, x1, y1, x2, y2, id)**.

Scroll bars can be placed using any dimensions, but the width of a vertical scroll bar, and the height of a horizontal scroll bar usually has a standard size. These can be determined by **scrollvertsiz[g](f, w, h)** and **scrollhorizsiz[g](f, w, h).**

A user movement of a scrollbar is given by the event **etsclpos**. This event does not move the scrollbar slider. This must be done by the program via **scrollpos(f, id, p)**.

When a user positions the scroll bar directly, it will follow the users mouse movements. However, it will return to the original position unless the **etsclpos** event is responded to and a **scrollpos(f, id, p)** call is made.

Besides position events, scrollbars issue two types of button pushes, referred to as line and page button events. The line buttons are the arrow buttons at either side of the scroll bar. The page buttons are the space between the scrollbar slider and the line arrows. A press anywhere in this area generates a page button event.

The page button area may not appear at all if the slider is fully to one side of the scrollbar.

The page and line terminology for these buttons occurs because their most common use is to scroll documents. In this case, the line buttons would be used to move the document one line up or down. In the case of horizontal scrollbars, this would be one character left or right (despite the term "line" in the buttons name). The page refers to one screenful of text, in any direction. If the page up button is hit, for example, the document would move one screenful up.

It's up to the program to implement the actions for line up/down and page up/down. In fact, these events can be used for any purpose in client programs.

Besides the position of the slider, its size can also be controlled by **scrollsiz(f, id, s)**.

The standard use of the scrollbar size is to indicate what proportion of the document or display is contained within the onscreen display, vs. the total size of the document. For example, if the document has 50% of its total displayed, then the size of the scrollbar is set to %50, which is done by:

```
scrollsiz(f, n, maxint div 2);
```

Scroll bars cannot be selected, enabled or disabled, or have face text read or written.

A number can be selected in an edit box by **numselbox[g](f, x1, y1, x2, y2, l, u, id)**.

The first number that appears in the number select box is by default the lower bound.

Number select boxes are an easy way to enter numbers from the user, and automatically restrict the input to numbers only. The numbers are entered in decimal, and cannot be negative. The edit box allows the number to be directly edited. Also, there are usually up and down buttons that allow the user to count the number up or down by one.

The size of a number select box is found by **numselboxsiz[g](f, l, u, w, h)**.

Number select boxes cannot be selected, enabled or disabled, or have face text read or written.

A general string can be edited with **editbox[g](f, x1, y1, x2, y2, id)**.

And empty edit box is placed, and the user has the ability to edit text into the box, with cursor movements, character delete, etc.

An edit box can be presented blank, or default text can be placed into the edit box. If the user presses enter to the box, it sends a **etedtbox** event. However, the program can use any method to signal done, such as a button next to the edit control. The resulting text can then be retrieved from the edit box.

The size of an edit box is found by **editboxsiz[g](f, s, w, h)**.

Edit boxes cannot be selected, enabled or disabled, or have face text read or written.

A list box is a series of items that can be selected. It is placed with **listbox[g](f, x1, y1, x2, y2, sp, id)**, where **sp** is a list of strings to display.

The string list definition appears as:

```
{ string set for list box }

strptr = ^strrec;
strrec = record

   next: strptr; { next entry in list }
   str:  pstring { string }

end;
```

The string pointer is a list of strings, each string of which describes an entry in the list box.

When the user selects an item from the list box, the **etlstbox** event is returned. This event gives the id of the widget, and the number of the select, from the top. The first item in the list will be 1, the second 2, etc.

The size of a list box is found by **listboxsiz[g](f, sp, w, h)**.

List boxes cannot be selected, enabled or disabled, or have face text read or written.

The same multiple string selection can be done in a different way by **dropbox[g](f, x1, y1, x2, y2, sp, id)**.

A drop box only shows the whole list if the user selects it, otherwise only the currently selected entry is shown. The full list "drops down" from the selection box when the user selects it. A drop box takes less space than a list box when it is not selected. A drop box selection is signaled by the **etdrpbox** event, which gives the widget id, and the number of the selection, from 1 to n.

The size of a drop box is found by **dropboxsiz[g](f, sp, cw, ch, ow, oh)**. Because drop boxes have two appearances, one when open, and one when closed, both sizes are returned. The closed appearance gives the basic size of the widget, but the open size makes is possible to determine if the list will go beyond the edge of the window when open.

Drop boxes cannot be selected, enabled or disabled, or have face text read or written.

Very similar to a drop box, a drop/edit box allows selection from a list, but also allows the current selection string to be edited.

A drop/edit box is placed with **dropeditbox[g](f, x1, y1, x2, y2, sp, id)**. When a selection is made from the drop/edit box, the **etdrebox** event is sent, which includes the widget identifier. The selection data itself is a string, and must be retrieved with **getwidgettext(f, id, s)**.

The size of a drop/edit box is found by **dropeditboxsiz[g]**. Because drop/edit boxes have two appearances, one when open, and one when closed, both sizes are returned. The closed appearance gives the basic size of the widget, but the open size makes is possible to determine if the list will go beyond the edge of the window when open.

Drop/edit boxes cannot be selected, enabled or disabled, or have face text read or written.

Sliders are linear controls that can be placed either horizontally or vertically.

A vertical slider is placed with **slidevert[g](f, x1, y1, x2, y2, m, id).** A horizontal slider can be placed by **slidehoriz[g](f, x1, y1, x2, y2, m, id).**

Sliders indicate changes in their position with the event **etsldpos**. This gives the widget id, and a **maxint** ratioed position of the slider, from 0 to **maxint**. 0 is the top or leftmost position of the slider, and **maxint** is the bottom or rightmost position of the slider.

The size of a slider can be determined by **scrollvertsiz[g](f, w, h)**.

Sliders cannot be selected, enabled or disabled, or have face text read or written.

Tab bars allow the user to select from a series of labeled tabs, usually to specify locations in document. A **tabbar** is a group box with tabs on one, two, three or four edges. The tabs can be placed on the top, bottom, left or right side of the included client area.

A tabbar is placed by **tabbar[g](f, x1, y1, x2, y2, spt, spr, spb, spl, id)**.

Tabbar selections are indicated by the event **ettabbar**, which gives the widget id, the side which generated the event and the tab number selection, from 1 to n, counting from the first string entry in the list.

The size of a tabbar is found by **tabbarsiz[g](f, tat, tar, tab, tal, cw, ch, w, h, ox, oy)**. A **tabbar** acts like a group box, and has a client area to place child windows or widgets. The required client size can be specified, and the sizing call returns the offset required to find the client location within the **tabbar**.

If the **tabbar** must fit into a fixed window size, the size of the resulting client for a **tabbar** can be found with **tabbarclient[g](f, tat, tar, tab, tal, w, h, cw, ch, ox, oy)**. This returns the client width and height, and its offset from the origin of the **tabbar**.

Tab bars cannot be selected, enabled or disabled, or have face text read or written.

## N.10 Components

A background box is placed by **background[g](f, x1, y1, x2, y2, id)**. A background box is designed to serve as the background to a series of controls, and it has the standard color for such backgrounds.

Background boxes have no sizing, because there are no borders or other content. They are just a colored rectangle. Background boxes cannot be selected, enabled or disabled, or have face text read or written.

A group box is similar to a background box, but it has a label for the "group" of controls contained within it. It is placed by **group[g]( f, x1, y1, x2, y2, id)**.

The size of a group box found by **groupsiz[g](f, s, w, h, ox, oy)**. A group box has a client area to place child windows or widgets. The required client size can be specified, and the sizing call returns the offset required to find the client location within the group box.

Group boxes cannot be selected, enabled or disabled, or have face text read or written.

A progress bar is used to indicate the progress of a job completion, like installing software, saving a file, etc.

It is placed by **progbar[g](f, x1, y1, x2, y2, id)**. The initial progress indication is zero when placed. The size of the progress bar is set by **progbarpos(f, id, pos)**.

The size of a progress bars can be determined by **progbarsiz[g](f, w, h)**.

Progress bars cannot be selected, enabled or disabled, or have face text read or written.

## N.11 Dialogs

A dialog is a completely separate window which is preformatted with widgets. Dialogs introduce complex queries into a program, using the look of the native operating system,.

Dialogs display a property known as modality. Since the dialog is a separate window, it can be independent of the other windows created by the calling task, or the dialog can be forced to appear at the top of the applications stacking order.

**widgets** uses two types of modality. If the task that created the dialog also created other windows, the dialog is forced to the front of the other windows, and selection of the other task windows is disabled. This indicates to the user that only the dialog is currently being managed by the task.

If windows are created by different threads, then the dialog will not be modal vs. the other thread's windows. This reflects the fact that the windows outside the dialog can run while the dialog does.

An alert dialog is used to send errors or other important messages to the user. It has a window title, a message that constitutes the alert, and typically has an "ok" or "close" button for the user to indicate they have seen it.

An alert is created by **alert(title, msg).** The alert call will not return until the user has clicked the ok button for the alert.

The query dialogs allow the user to select important information such as a file, a search string, or color or a font. They use a model called "flow through". The query may select several types of information, and will accept a default setting, allow the user to select a new setting, and return that. The flow through model means that several parameters are set up before the call, may or may not be modified by the query, then are returned to the caller. The calling thread stops until the dialog is completed by the user.

All of the parameters of a dialog may or may not be implemented in the actual system dialog. The flowthrough system allows for differences in systems. Since the variables are preinitalized with the defaults, if the dialog does not implement a particular parameter, the value will be left unchanged.

A color can be chosen by **querycolor(r, g, b).** The default color is set before the call, and the possibly changed color is returned by the call.

A file to open name is selected by **queryopen(s)**.

The default filename is passed in as a dynamic string, and a different string is returned as the result of the dialog. The string returned is different than the input string, and the program is responsible for

disposing of both strings. If the dialog is canceled instead of completed by the user, the string returned is zero length. This means to not proceed with the open operation.

A file to save name is selected by **querysave(s)**.

The default filename is passed in as a dynamic string, and a different string is returned as the result of the dialog. The string returned is different than the input string, and the program is responsible for disposing of both strings. If the dialog is canceled instead of completed by the user, the string returned is zero length. This means to not proceed with the save operation.

A string to search for is selected by **queryfind(s, opt)**.

The option flags are given by a set of flags:

```
{ settable items in find query }
qfnopt = (qfncase, { Case sensitive }
          qfnup},  { Search up/Search down }
          qfbre);
qfnopts = set of qfnopt;
```

The default search string is passed in as a dynamic string. The string returned is different than the input string, and the program is responsible for disposing of both strings. If the dialog is canceled instead of completed by the user, the string returned is zero length. This means to not proceed with the search operation.

A string to search for and replace is selected by **queryfindrep(s, r, opt)**.

The option flags are given by a set of flags:

```
{ settable items in replace query }
qfropt = (qfrcase, { case sensitive }
          qfrup,   { search up/search down }
          qfrfind,
          qfrallfil,
          qfralllin);
qfropts = set of qfropt;
```

The default search string and replacement strings are passed in as a dynamic strings. The strings returned are different than the input string, and the program is responsible for disposing of all strings. If the dialog is canceled instead of completed by the user, the strings returned are zero length. This means to not proceed with the search/replace operation.

Fonts are selected by **queryfont(f, fc, s, fr, fg, fb, br, bg, bb, effect)**.

The font effects are declared as:

```
{ effects in font query }
qfteffect = (qfteblink, qftereverse, qfteunderline,
             qftesuperscript, qftesubscript, qfteitalic,
             qftebold, qftestrikeout, qftestandout,
             qftecondensed, qfteextended, qftexlight,
```

```
              qftelight, qftexbold, qftehollow, qfteraised);
qfteffects = set of qfteffect;
```

## N.12 Events

The definition of an event record is upward compatible with previous event record declarations from **terminal**, **graphics** and **windows**.

```
module widgets;

{ events }
evtcod = (etchar,     { ANSI character returned }
          etup,       { cursor up one line }
          etdown,     { down one line }
          etleft,     { left one character }
          etright,    { right one character }
          etleftw,    { left one word }
          etrightw,   { right one word }
          ethome,     { home of document }
          ethomes,    { home of screen }
          ethomel,    { home of line }
          etend,      { end of document }
          etends,     { end of screen }
          etendl,     { end of line }
          etscrl,     { scroll left one character }
          etscrr,     { scroll right one character }
          etscru,     { scroll up one line }
          etscrd,     { scroll down one line }
          etpagd,     { page down }
          etpagu,     { page up }
          ettab,      { tab }
          etenter,    { enter line }
          etinsert,   { insert block }
          etinsertl,  { insert line }
          etinsertt,  { insert toggle }
          etdel,      { delete block }
          etdell,     { delete line }
          etdelcf,    { delete character forward }
          etdelcb,    { delete character backward }
          etcopy,     { copy block }
          etcopyl,    { copy line }
          etcan,      { cancel current operation }
          etstop,     { stop current operation }
          etcont,     { continue current operation }
          etprint,    { print document }
          etprintb,   { print block }
          etprints,   { print screen }
          etfun,      { function key }
          etmenu,     { display menu }
          etmouba,    { mouse button assertion }
          etmoubd,    { mouse button deassertion }
          etmoumov,   { mouse move }
          ettim,      { timer matures }
          etjoyba,    { joystick button assertion }
          etjoybd,    { joystick button deassertion }
          etjoymov,   { joystick move }
          etterm,     { terminate program }
```

```
            etmoumovg,  { mouse move graphical }
            etframe,    { frame sync }
            etresize,   { window was resized }
            etredraw,   { window redraw }
            etmin,      { window minimized }
            etmax,      { window maximized }
            etnorm,     { window normalized }
            etmenus,    { menu item selected }
            etbutton,   { button assert }
            etchkbox,   { checkbox click }
            etradbut,   { radio button click }
            etsclull,   { scroll up/left line }
            etscldrl,   { scroll down/right line }
            etsclulp,   { scroll up/left page }
            etscldrp,   { scroll down/right page }
            etsclpos,   { scroll bar position }
            etedtbox,   { edit box signals done }
            etnumbox,   { number select box signals done }
            etlstbox,   { list box selection }
            etdrpbox,   { drop box selection }
            etdrebox,   { drop edit box selection }
            etsldpos,   { slider position }
            ettabbar,   { tab bar select }
  { event record }
  evtrec = record

     winid: ss_filhdl; { identifier of window for event }
     case etype: evtcod of { event type }

        { ANSI character returned }
        etchar:   (char:                char);
        { timer handle that matured }
        ettim:     (timnum:  timhan);
        etmoumov:  (mmoun:   mouhan;   { mouse number }
                    moupx,
                    moupy:   integer); { mouse movement }
        etmouba:   (amoun:   mouhan;   { mouse handle }
                    amoubn:  moubut);  { button number }
        etmoubd:   (dmoun:   mouhan;   { mouse handle }
                    dmoubn:  moubut);  { button number }
        etjoyba:   (ajoyn:   joyhan;   { joystick number }
                    ajoybn:  joybut);  { button number }
        etjoybd:   (djoyn:   joyhan;   { joystick number }
                    djoybn:  joybut);  { button number }
        etjoymov:  (mjoyn:   joyhan;   { joystick number }
                    joypx,
                    joypy,
                    joypz:   integer); { joystick coordinates }
        etfun:     (fkey:    funky);   { function key }
```

```
      etmoumovg: (mmoung:  mouhan;   { mouse number }
                  moupxg,
                  moupyg:  integer); { mouse movement }
      etredraw:  (rsx,
                  rsy,
                  rex,
                  rey:     integer); { redraw screen }
      etmenus:   (menuid:  integer); { menu item selected }
      etbutton:  (butid:   integer); { button id }
      etchkbox:  (ckbxid:  integer); { checkbox }
      etradbut:  (radbid:  integer); { radio button }
      etsclull:  (sclulid: integer); { scroll up/left line }
      etscldrl:  (scldlid: integer); { scroll down/right line}
      etsclulp: (sclupid:  integer); { scroll up/left page }
      etscldrp:  (scldpid: integer); { scroll dwn/rgt page }
      etsclpos:  (sclpid:  integer;  { scroll bar }
                  sclpos:  integer); { scroll bar position }
      etedtbox:  (edtbid:  integer); { edit box complete }
      etnumbox:  (numbid:  integer;  { num sel box select }
                  numbsl:  integer); { num select value }
      etlstbox:  (lstbid:  integer;  { list box select }
                  lstbsl:  integer); { list box select num }
      etdrpbox:  (drpbid:  integer;  { drop box select }
                  drpbsl:  integer); { drop box select }
      etdrebox:  (drebid:  integer); { drop edit box select }
      etsldpos:  (sldpid:  integer;  { slider position }
                  sldpos:  integer); { slider position }
      ettabbar:  (tabid:   integer;  { tab bar }
                  tabor:   tabori;   { tab side }
                  tabsel:  integer); { tab select }
      etup, etdown, etleft, etright, etleftw, etrightw,
      ethome, ethomes, ethomel, etend, etends, etendl, etscrl,
      etscrr, etscru, etscrd,etpagd, etpagu, ettab, etenter,
      etinsert, etinsertl, etinsertt, etdel, etdell, etdelcf,
      etdelcb, etcopy, etcopyl, etcan, etstop, etcont,
      etprint, etprintb, etprints, etmenu, etterm, etframe,
      etresize, etmin, etmax, etnorm, et_fndtrm, et_wigstr,
      et_winstr, et_wincls: (); { normal events }

   { end }

end;
```

## N.13 Event callbacks

As in **terminal, graphics** and **windows**, the events in **widgets** also be accessed via a series of **virtual** procedures:

```
module widgets;

virtual procedure evbutton(id: integer); begin end;
virtual procedure evchkbox(id: integer); begin end;
virtual procedure evradbut(id: integer); begin end;
virtual procedure evsclull(id: integer); begin end;
virtual procedure evscldrl(id: integer); begin end;
virtual procedure evsclulp(id: integer); begin end;
virtual procedure evscldrp(id: integer); begin end;
virtual procedure evsclpos(id: integer; pos: integer); begin end;
virtual procedure evedtbox(id: integer); begin end;
virtual procedure evnumbox(id: integer; sl: integer); begin end;
virtual procedure evlstbox(id: integer; sl: integer); begin end;
virtual procedure evdrpbox(id: integer; sl: integer); begin end;
virtual procedure evdrebox(id: integer); begin end;
virtual procedure evsldpos(id: integer; pos: integer); begin end;
virtual procedure evtabbar(id: integer; tor: tabori; sel: integer);
   begin end;

begin !
end.
```

## N.14 Widget Classes

The functions in **widgets** are also available in a series of classes to enable object oriented design. The methods used to define widgets as objects is different than, but similar to, the procedure oriented calls outlined previously. The methods used are reshaped to be in a form more suited for object oriented design. Not all widgets are appropriate as objects. Only controls and components are so defined. Dialogs are left to procedural interfaces.

Note that the events generated by class based widgets are the same values and formats as their procedural versions.

The base class for widgets is:

```
class widget(input, parent: text; id: integer);

procedure select(e: boolean); begin end;
procedure enable(e: boolean); begin end;
procedure gettext(var s: pstring); begin end;
procedure puttext(view s: string); begin end;
procedure pos(x, y: integer); begin end;
procedure size(w, h: integer); begin end;
procedure minsiz(var w, h: integer); begin end;
procedure show; begin end;

.
```

Logically, a widget is a derived class of the **childwindow** class in **windows**. As in the case of **childwindow**, the parent window is specified as a parameter to the class, and the widget class carries

out its own event management. However, unlike a true derived class, widgets do not allow access to the base **childwindow** class. Because of this, widget classes also require the input file to be specified. This indicates where the widget events will be sent.

To allow a widget class to generate procedural events, the logical identifier number of the widget is specified. However, each widget class gives one or more callbacks for each of the events it generates. If all of the widget events are so specified, then the identifier given to the widget will not be used, and can be any value (for example 0).

A widget is created, but not drawn on screen or buffer, when it is instantiated. The parameter to the class is the text file that denotes the window the widget is to be drawn in. The widget is not drawn onscreen until its show method is executed. When the widget is deallocated, it is removed from the drawing surface. This characteristic of widget classes allows them to be set to the proper location, size, etc.

Some widgets can be selected, which changes their appearance to the select state. A widget is selected by the **select(e)** method.

Widgets can be enabled or disabled by the **enable(e)** method**.** Widgets are disabled by default. When a widget is disabled, it has the disabled appearance, such as greyed out.

Some widgets have text that can be set, read, or in some cases, both. The text in a widget is set by the **puttext(s)** method. The text in a widget can be read by the **gettext(s)** method. The default face text for a widget is an empty string.

The position of a widget can be set by the **pos(x, y)** method. The size of a widget can be set by the **size(w, h)** method. Both the position and size of a widget must be set before the widget is visible.

The minimum size of a widget can be found with the **minsiz(w, h)** method.

The **show** method causes the widget to be presented with the attributes that were set with the other methods.

Not all methods are used with all widgets. If a method is unimplemented for a particular class, it throws an exception.

The class names for widgets is derived from the procedural name for the same widget, but with a “c” appended. This prevents collisions with the procedural names.

The classes that are derived from the widget class are:

```
class buttonc;

extends widget;

virtual procedure pressed; begin end;

.
```

```
class buttongc;

extends widget;

virtual procedure pressed; begin end;

.
```

Presents a button.

The minimum size is based on the face text, so the face text should be placed into the object before the **minsiz** method is called.

If the button is pressed, the event will be sent to the **pressed** method.

```
class checkboxc;

extends widget;

virtual procedure selected; begin end;

.

class checkboxgc;

extends widget;

virtual procedure selected; begin end;

.
```

Presents a checkbox.

The minimum size is based on the face text, so the face text should be placed into the object before the **minsiz** method is called.

If the checkbox is selected, the event will be sent to the **selected** method.

```
class radiobuttonc;

extends widget;

virtual procedure selected; begin end;

.
```

```
class radiobuttongc;

extends widget;

virtual procedure selected; begin end;

.
```

Presents a radio button.

The minimum size is based on the face text, so the face text should be placed into the object before the **minsiz** method is called.

If the checkbox is selected, the event will be sent to the **selected** method.

```
class groupc;

extends widget;

.

class groupgc;

extends widget;

.
```

Presents a group box.

The minimum size is based on the face text, so the face text should be placed into the object before the **minsiz** method is called.

```
class backgroundc;

extends widget;

.

class backgroundgc;

extends widget;

.
```

Presents a background box. The **gettext** and **puttext** methods are inoperative on a background box.

```
class scrollvertc;

extends widget;

procedure sliderpos(p:integer); begin end;
procedure slidersiz(s:integer); begin end;

virtual procedure upline; begin end;
virtual procedure uppage; begin end;
virtual procedure downline; begin end;
virtual procedure downpage; begin end;
virtual procedure newpos(pos: integer); begin end;

.

class scrollvertgc;

extends widget;

procedure sliderpos(p:integer); begin end;
procedure slidersiz(s:integer); begin end;

virtual procedure upline; begin end;
virtual procedure uppage; begin end;
virtual procedure downline; begin end;
virtual procedure downpage; begin end;
virtual procedure newpos(pos: integer); begin end;

.
```

Presents a vertical scrollbar. The **gettext** and **puttext** methods are inoperative on a vertical scroll bar. The method **sliderpos** controls the position ratioed to **maxint**. The method **slidersiz** controls the size of the slider ratioed to **maxint**.

If the up one line button on the scrollbar is activated, the event is sent to the **upline** method. Similarly, up one page is sent to **uppage**, down one line is sent to **downline**, and down one page is sent to **downpage**.

If the slider position is changed, the **newpos** event is sent, which includes a maxint ratioed position for the slider, where 0 means top, and **maxint** means bottom.

```
class scrollhorizc;

extends widget

procedure sliderpos(p:integer); begin end;
procedure slidersiz(s:integer); begin end

virtual procedure leftline; begin end;
virtual procedure leftpage; begin end;
virtual procedure rightline; begin end;
virtual procedure rightpage; begin end;
virtual procedure newpos(pos: integer); begin end;

.

class scrollhorizgc;

extends widget

procedure sliderpos(p:integer); begin end;
procedure slidersiz(s:integer); begin end

virtual procedure leftline; begin end;
virtual procedure leftpage; begin end;
virtual procedure rightline; begin end;
virtual procedure rightpage; begin end;
virtual procedure newpos(pos: integer); begin end;

.
```

Presents a horizontal scrollbar. The **gettext** and **puttext** methods are inoperative on a horizontal scroll bar. The method **sliderpos** controls the position ratioed to **maxint**. The method **slidersiz** controls the size of the slider ratioed to **maxint**.

If the left one line button on the scrollbar is activated, the event is sent to the **leftline** method. Similarly, left one page is sent to **leftpage**, right one line is sent to **rightline**, and right one page is sent to **rightpage**.

If the slider position is changed, the **newpos** event is sent, which includes a maxint ratioed position for the slider, where 0 means top, and **maxint** means bottom.

```
class numselboxc(l, u: integer);

extends widget

virtual procedure select(sl: integer); begin end;

.

class numselboxgc(l, u: integer);

extends widget

virtual procedure select(sl: integer); begin end;

.
```

Presents a number select box with the lower limit **l** and the upper limit **u**.

If a number is selected, the event is sent to the **select** method with the number selected in **sl**.

```
class editboxc;

extends widget

virtual procedure enter; begin end;

.

class editboxgc;

extends widget

virtual procedure enter; begin end;

.
```

Presents and edit box. When sizing an edit box, the face text of the widget should be set to a "representative" string, which is then used to set the minimum size of the box. This can be either the default text that will be placed into the edit box to start, or a string representing the maximum string width that would be encountered. For example, "WWWWWWWW" (8 "W" characters) would be the maximum size for 8 characters, since "W" is the widest character. After setting the minimum size, the face text can be changed, or set to blank.

When the text in the **editbox** is entered, the event is sent to the **enter** method. This means that the resulting text can be retrieved via the **gettext** method.

```
class progbarc;

extends widget

procedure setpos(p: integer); begin end;

.

class progbargc;

extends widget

procedure setpos(p: integer); begin end;

.
```

Presents a progress bar. The current position of the progress bar can be set via the **setpos** method, which takes a position from 0 to **maxint**, where 0 is “nothing done”, and **maxint** is “all done”.

```
class listboxc(sp: strptr);

extends widget

virtual procedure select(sl: integer); begin end;

.

class listboxgc(sp: strptr);

extends widget

virtual procedure select(sl: integer); begin end;

.
```

Presents a **listbox**. The **listbox** takes a list of strings to be selected. When the user selects one of the strings, an event is sent that contains the number of the string in **sl**, which is 1 to n in order of the string list used to create the **listbox**.

```
class dropboxc(sp: strptr);

extends widget

virtual procedure select(sl: integer); begin end;

.
```

```
class dropboxgc(sp: strptr);

extends widget

virtual procedure select(sl: integer); begin end;

.
```

Presents a **dropbox**. The **dropbox** takes a list of strings to be selected. When the user selects one of the strings, an event is sent that contains the number of the string in **sl**, which is 1 to n in order of the string list used to create the **dropbox**.

```
class slidehorizc;

extends widget

virtual procedure sldpos(p: integer); begin end;

.

class slidehorizgc;

extends widget

virtual procedure sldpos(p: integer); begin end;

.
```

Presents a horizontal slider. When the slider is moved, it generates an event procedurally as **etsldpos** or as a callback **newpos**. The position in **sldpos** is from 0 to **maxint**, where 0 is the extreme left, and **maxint** is the extreme right.

```
class slidevertc;

extends widget

virtual procedure sldpos(p: integer); begin end;

.

class slidevertgc;

extends widget

virtual procedure sldpos(p: integer); begin end;

.
```

Presents a vertical slider. When the slider is moved, it generates an event procedurally as **etsldpos** or as a callback **newpos**. The position in **sldpos** is from 0 to **maxint**, where 0 is the extreme top, and **maxint** is the extreme bottom.

```
class tabbarc(spt, spr, spb, spl: strptr);

extends widget

virtual procedure select(ori: tabori; sel: integer); begin end;

.
```

Presents a **tabbar**. The **tabbar** is presented using a list of strings for each of the sides, top, right, bottom and left, corresponding to **spt**, **spr**, **spb**, and **spl**, respectively. When a tab is selected, it sends a procedural event **ettabbar**, or a callback **select**. The orientation of the tab is given in **ori**, and the number of the string in the tab list is **sel**, which is 1 to n in the order of the string list.

## N.15 exceptions

The following exceptions are generated in **widgets**:

| Identifier | Meaning |
|---|---|
| **WidgetNotFound** | Widget by logical ID not found |
| **WidgetIdDuplicate** | Widget logical ID was duplicated |
| **CannotSelectWidget** | Widget is not selectable |
| **CannotPutTextWidget** | Cannot put text in this widget |
| **CannotGetTextWidget** | Cannot get text from this widget |
| **CannotDisableWidget** | Cannot disable this widget |
| **InvalidScrollBarPosition** | Invalid scroll bar position specified |
| **InvalidScrollBarSize** | Invalid scroll bar size specified |
| **ControlCreateFail** | Attempt to create control fails |
| **InvalidProgressBarPosition** | Invalid progress bar position specified |
| **NoStringSpace** | Out of string space |
| **UnableCreateTab** | Unable to create tab in tab bar |
| **UnableCreateFileDialog** | Unable to create file dialog |
| **UnableCreateFindDialog** | Unable to create find dialog |
| **UnableCreateFontDialog** | Unable to create font dialog |
| **StringTooLong** | String to long for find/replace dialog |
| **InvalidTabSelect** | Logical tab to select was invalid |

**widgets** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **widgets** will go back to **widgets**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **widgets** need only capture the exceptions occurring in the procedure or function that is called.

## N.16 Procedures and functions in widgets

For all of the following calls, If the screen file **f** is not present, the default is the standard **output** file. If the procedure or function is a method, the output screen file should not be specified, since it is inherent in the object.

procedure killwidget([var f: text;] id: integer);

The widget within window **f** with the logical identifier **id** is removed from the system. It will be erased from the screen, and contents under it will be restored as required.

Exceptions: **WidgetNotFound**

procedure selectwidget([var f: text; id: integer;] e: boolean);

The widget within window **f** by the logical identifier **id** will enter the select state if **e** is true, otherwise the select state is removed. The exact effect of the select state depends on the widget. See the individual widget to be selected for more information. It is an error to select a widget that has no selectability.

Selection is typically used to indicate that the widget is "on", by changing its face appearance. For example, it can be checked, pressed or a similar visual state change.

Exceptions: **WidgetNotFound, CannotSelectWidget**

procedure enablewidget([var f: text;] id: integer; e: boolean);

The widget within window **f** by the logical identifier **id** will enter the enable state if **e** is true, otherwise the disable state is entered. The exact effect of the enable or disable state depends on the widget. See the individual widget to be selected for more information. It is an error to enable or disable a widget that has no such capability. The default state for all widgets is to be enabled. A widget that is disabled will stop sending events.

Disabling a widget is typically used to mark it as unusable or invalid in the current context. An example would be a "next" button where no next page or item exists. The standard method used to indicate disabled widgets is to give them a "greyed out" appearance, but the actual effect may depend on the operating system.

Exceptions: **WidgetNotFound**, **CannotDisableWidget**

procedure getwidgettext([var f: text;] id: integer; var s: pstring);

Retrieves the text contained by the widget with the logical identifier **id** within window **f**, and returns the text as a dynamic string **s**. It depends on the widget as to if it has text that can be read. It is an error to get text from a widget that has no such capability.

Widgets can typically have their text read if the widget provides the user with the ability to modify or edit text. In this case, retrieving the text is required to obtain the new text.

Exceptions: **WidgetNotFound, CannotGetTextWidget**

procedure putwidgettext([var f: text;] id: integer; view s: string);

Places text in the widget with the logical identifier **id** within window **f** from the string **s**. It depends on the widget as to if it can accept text placed in this manner. It is an error to place text in a widget that has no such capability.

A widget will have the ability to place text if it can edit text by the user. Placing text in the widget can be used to initialize the contents of such a widget, or as part of the overall interaction with the user.

Exceptions: **WidgetNotFound, CannotPutTextWidget**

procedure sizwidget[g]([var f: text;] id: integer; x, y: integer);

Resize an existing widget. The widget with the logical identifier **id** is resized to be the size in **x** and **y**, in the window **f**.

If the graphical version is used, the size is in pixels, otherwise, the size is in characters.

Exceptions: **WidgetNotFound**

procedure poswidget[g]([var f: text;] id: integer; x, y: integer);

Reposition an existing widget. The widget with the logical identifier **id** is repositioned to be at the position **x** and **y**, in the parent window of the window f.

If the graphical version is used, the size is in pixels, otherwise, the size is in characters.

Exceptions: **WidgetNotFound**

procedure buttonsiz[g]([var f: text;] view s: string; var w, h: integer);

Finds the minimum size of a button within window **f**, with face text string **s**. The width to use is returned in **w**, and the height in **h**. Button sizing returns the minimum size in terms of the minimum amount of space needed to contain the face text, in the labeling font, and the border of the button itself. You will want to use the size as a guide to button sizing, and not for the actual size of the button. A good rule of thumb is to add %25 of the minimum height of the button to the actual height and width of the button to give the user sufficient area to click on the button. In addition, buttons should be "justified" when they appear in groups to appear as the same width. This is done by calculating a maximum size for a group of related buttons, then using the same width and height for all of them.

Exceptions: None

procedure button[g]([var f: text;] x1, y1, x2, y2: integer; view s: string; id: integer);

Places a button within window **f** in the bounding box formed by **x1**, **y1**, **x2**, **y2**, with the face text from the string **s**. The logical identifier is specified in **id**, which must be an integer from 1 to n, which is not currently in use in any other widget. The button drawn is not guaranteed to completely fill the bounding box. The button should be placed against a background that is colored with the standard background color. The button will be placed at the front of the window stacking order, and should be placed after any other widgets or child windows that the button is to appear in front of.

When the button is pressed, it will send an **etbutton** event, which contains the logical id of the button that was pressed. It is up to the system whether the event occurs when the button is depressed or released.

Buttons cannot be selected, or have their face text changed or read. Buttons can be enabled and disabled. If the button is disabled, it will not sent **etbutton** events when pressed.

Exceptions: **WidgetIdDuplicate**

procedure checkboxsiz[g]([var f: text;] view s: string; var w, h: integer);

Finds the minimum size of a checkbox within window **f**, with face text string **s**. The width to use is returned in **w**, and the height in **h**. Checkbox sizing returns the minimum size in terms of the minimum amount of space needed to contain the face text, in the labeling font, and the border of the checkbox itself, if it exists. You will want to use the size as a guide to checkbox sizing, and not for the actual size of the checkbox. A good rule of thumb is to add %25 of the minimum height of the checkbox to the actual height and width of the checkbox to give the user sufficient area to click on the checkbox.

Exceptions: None

procedure checkbox[g]([var f: text;] x1, y1, x2, y2: integer; view s: string; id: integer);

Places a checkbox within window **f** in the bounding box formed by **x1**, **y1**, **x2**, **y2**, with the face text from the string **s**. The logical identifier is specified in **id**, which must be an integer from 1 to n, which is not currently in use in any other widget. The checkbox drawn is not guaranteed to completely fill the bounding box. The checkbox should be placed against a background that is colored with the standard background color. The checkbox will be placed at the front of the window stacking order, and should be placed after any other widgets or child windows that the checkbox is to appear in front of.

When the checkbox is clicked, it will send an **etchkbox** event, which contains the logical id of the checkbox that was pressed. It is up to the system whether the event occurs when the checkbox is depressed or released.

Checkboxes cannot have their face text changed or read. A checkbox can be selected or deselected, and can be enabled and disabled. If the checkbox is selected, it will appear with a selected face, which is typically a checkmark in a box. If the checkbox is disabled, it will not sent **etchkbox** events when pressed.

A checkbox only changes its appearance in response to a select, and does not keep a state that can be read by the program. It's up to the program to keep track of the state of the checkbox, and how to handle it. In particular, if the **checkbox** is pressed, it is up to the program to change its select status, otherwise the press will have no effect. The program can implement many different effects for checkboxes. The **checkbox** can toggle, or it can be one of a series of mutually exclusive selections.

Exceptions: **WidgetIdDuplicate**

procedure radiobuttonsiz[g]([var f: text;] view s: string; var w, h: integer);

Finds the minimum size of a radio button within window **f**, with face text string **s**. The width to use is returned in **w**, and the height in **h**. Radio button sizing returns the minimum size in terms of the minimum amount of space needed to contain the face text, in the labeling font, and the border of the radio button, if it exists. You will want to use the size as a guide to radio button sizing, and not for the actual size of the checkbox. A good rule of thumb is to add %25 of the minimum height of the radio button to the actual height and width of the radio button to give the user sufficient area to click on the radio button.

Exceptions: None

procedure radiobutton[g]([var f: text;] x1, y1, x2, y2: integer; view s: string; id: integer);

Places a radio button within window **f** in the bounding box formed by **x1**, **y1**, **x2**, **y2**, with the face text from the string **s**. The logical identifier is specified in **id**, which must be an integer from 1 to n, which is not currently in use in any other widget. The radio button drawn is not guaranteed to completely fill the bounding box. The radio button should be placed against a background that is colored with the standard background color. The radio button will be placed at the front of the window stacking order, and should be placed after any other widgets or child windows that the radio button is to appear in front of.

When the radio button is pressed, it will send an **etradbut** event, which contains the logical id of the radio button that was pressed. It is up to the system whether the event occurs when the radio button is depressed or released.

Radio buttons cannot have their face text changed or read. A radio button can be selected or deselected, and can be enabled and disabled. If the radio button is selected, it will appear with a selected face, which is typically a blacked out radio button. If the radio button is disabled, it will not sent **etradbut** events when pressed.

A radio button only changes its appearance in response to a select, and does not keep a state that can be read by the program. It's up to the program to keep track of the state of the radio button, and how to handle it. In particular, if the checkbox is pressed, it is up to the program to change its select status, otherwise the press will have no effect. The program can implement many different effects for radio buttons. The radio button can toggle, or it can be one of a series of mutually exclusive selections.

Exceptions: **WidgetIdDuplicate**

procedure groupsiz[g]([var f: text;] view s: string; cw, ch: integer; var w, h, ox, oy: integer);

Finds the required size of a group box in window **f**, with the face text given in string **s**, and the client area width **cw**, and client area height **ch**. The required width is returned in **w**, the height in **h**, and the offset to the client area in **x** and **y**. A group box consists of a border area, a label, and an internal client area. Group boxes are designed to be layered components. They contain other widgets, and provide a background for them. When a group size is found, the minimum size is found as what will contain all of the border, face text and the requested client area. The program will know where to place its widgets in the client area by the client offset, which is given as a difference between the bounding box origin, and the origin of the client rectangle.

Exceptions: None

procedure group[g]([var f: text;] x1, y1, x2, y2: integer; view s: string; id: integer);

Creates a group box in window **f** within the bounding rectangle **x1**, **y1**, **x2**, **y2**, the face text given in string **s**, and with the logical identifier **id**. Group boxes are containers for other widgets, and consist of a border area, the face text, and an internal client area where other widgets are to be placed.

The entire client area of the group will have the standard background color, and widgets can be placed into the client in any arrangement or number. The program should be sure to create the client area widgets after the group is created, so that they will appear in front of the group in stacking order.

The location of the client area within a group box can be found with the group sizing call.

Exceptions: **WidgetIdDuplicate**

procedure background[g]([var f: text;] x1, y1, x2, y2: integer; id: integer);

Creates a background box in the window **f**, with the bounding rectangle **x1**, **y1**, **x2**, **y2**, and the logical widget identifier **id**. A background is simply a rectangle with the standard background color. It is more convenient than simply painting a rectangle on the window with the background color because it handles its own redraws. Because a background box has no borders, widgets can be placed within it anywhere, and in any number. Any widgets to be placed within the group should be created after the group box is created, so that that they are on top of the group box in stacking order.

Exceptions: **WidgetIdDuplicate**

procedure scrollvertsiz[g]([var f: text;] var w, h: integer);

Finds the size for a vertical scroll bar in window **f**. Returns the width in **w**, and the height in **h**. Scrollbars typically can be sized to any size, and the width of a vertical scroll bar is a suggested width designed to match others used in the same system. The height of a vertical scrollbar is simply a suggestion, and can be ignored.

If a scrollbar cannot be arbitrarily sized, then the width and height will reflect the dimensions of a fixed scrollbar.

Exceptions: None

procedure scrollvert[g]([var f: text;] x1, y1, x2, y2: integer; id: integer);

Creates a vertical scrollbar in window **f**, with bounding rectangle **x1**, **y1**, **x2**, **y2**, and logical widget identifier **id**. If possible, the scrollbar is made to fill the width and height requested. If this is not possible, then the largest scrollbar is created that fits within the bounding rectangle. There is no guarantee that the scrollbar will completely fill the rectangle. The area under the scrollbar should be drawn with the standard background color.

The scrollbar will generate several events when clicked. The **etsclull** event indicates the line up button of the scrollbar was pressed. The **etscldrl** event indicates the line down button of the scrollbar was pressed. The **etsclulp** event indicates the page up section of the scrollbar was pressed. The **etscldrp** event indicates the page down section of the scrollbar was pressed. It is system dependent as to whether the buttons generate their events on a button press or a button release.

The **etsclpos** event gives the position of the top of the slider after the user moves it. The position is returned as a ratioed **maxint** number, where 0 means the slider is at the top, and **maxint** means the slider is at the bottom. The number is affected by the size of the slider. If, for example, the slider occupies %50 of the scrollbar, then only the positions 0 to **maxint div** 2 will be generated. It is undefined as to exactly when **etsclpos** events occur. They may only be generated when the slider is moved and then released, or they may be generated continuously as the slider is moved. If the generation is continuous, then the movements are usually subject to "rate limiting" to keep them from generating events too fast to handle.

The scrollbar will not change position on its own. The scrollbar will generate events, and it is up to the program to use those to set the scrollbar slider position, and to take action on them, such as move the screen data up or down. The page up/down and line up/down terminology is suggestive of the use of these events, but it is up to the program exactly how to use or implement these functions. Typically, these are used to move the displayed area of a document one line up or down, and one page or screenful up or down.

The size of the scrollbar slider is set by default to small, but convenient for the user to press and manipulate. This can be left alone for programs that don't require sized scrollbar sliders. The size of the slider is set by **scrollsiz[g]**. Typically, it is used to set the ratio of onscreen data shown to the entire document or other data available. For example, if %50 of the document is being displayed, then the slider should occupy %50 of the scrollbar.

Exceptions: **WidgetIdDuplicate**

procedure scrollhorizsiz[g]([var f: text;] var w, h: integer);

Finds the size for a horizontal scroll bar in window **f**. Returns the width in **w**, and the height in **h**. Scrollbars typically can be sized to any size, and the height of a horizontal scroll bar is a suggested width designed to match others used in the same system. The width of a horizontal scrollbar is simply a suggestion, and can be ignored.

If a scrollbar cannot be arbitrarily sized, then the width and height will reflect the dimensions of a fixed scrollbar.

Exceptions: None

procedure scrollhoriz[g]([var f: text;] x1, y1, x2, y2: integer; id: integer);

Creates a horizontal scrollbar in window **f**, with bounding rectangle **x1**, **y1**, **x2**, **y2**, and logical widget identifier **id**. If possible, the scrollbar is made to fill the width and height requested. If this is not possible, then the largest scrollbar is created that fits within the bounding rectangle. There is no guarantee that the scrollbar will completely fill the rectangle. The area under the scrollbar should be drawn with the standard background color.

The scrollbar will generate several events when clicked. The **etsclull** event indicates the line left button of the scrollbar was pressed. The **etscldrl** event indicates the line right button of the scrollbar was pressed. The **etsclulp** event indicates the page left section of the scrollbar was pressed. The **etscldrp** event indicates the page right section of the scrollbar was pressed. It is system dependent as to whether the buttons generate their events on a button press or a button release.

The **etsclpos** event gives the position of the left of the slider after the user moves it. The position is returned as a ratioed **maxint** number, where 0 means the slider is at the left, and **maxint** means the slider is at the right. The number is affected by the size of the slider. If, for example, the slider occupies %50 of the scrollbar, then only the positions 0 to **maxint div** 2 will be generated. It is undefined as to exactly when **etsclpos** events occur. They may only be generated when the slider is moved and then released, or they may be generated continuously as the slider is moved. If the generation is continuous, then the movements are usually subject to "rate limiting" to keep them from generating events too fast to handle.

The scrollbar will not change position on its own. The scrollbar will generate events, and it is up to the program to use those to set the scrollbar slider position, and to take action on them, such as move the screen data left or right. The page left/right and line left/right terminology is suggestive of the use of these events, but it is up to the program exactly how to use or implement these functions. Typically, these are used to move the displayed area of a document one character left or right, and one page or screenful left or right.

The size of the scrollbar slider is set by default to small, but convenient for the user to press and manipulate. This can be left alone for programs that don't require sized scrollbar sliders. The size of the slider is set by **scrollsiz[g]**. Typically, it is used to set the ratio of onscreen data shown to the entire document or other data available. For example, if %50 of the document is being displayed, then the slider should occupy %50 of the scrollbar.

Exceptions: **WidgetIdDuplicate**

procedure scrollpos([var f: text;] id: integer; p: integer);

Sets the scrollbar slider position for window **f**, scrollbar identifier **id**, to position **p**. The position is in ratioed **maxint** format. That is, 0 means to set the position to the top or left, and **maxint** means bottom or right. The position is affected by the size of the scrollbar slider. For example, if the slider occupies %50 of the scrollbar, then the range of positions would only be from 0 to **maxint** div 2. If the position given is beyond the maximum position possible, then the slider is set to the maximum travel position, and no error occurs. It is an error if the position is negative.

The program must specifically set the position of the scrollbar. The user moving the scrollbar slider may temporarily move the slider while it is being moved, but this will not remain in position after the user releases it. The program must specifically set the scrollbar position in response to the event.

Exceptions: **WidgetNotFound, InvalidScrollBarPosition**

procedure scrollsiz([var f: text;] id: integer; s: integer);

Sets the size of the scrollbar slider in window **f**, logical identifier **id**, to the size **s**. The size of the scrollbar slider is a **maxint** ratio, with 0 meaning infinitely small, and **maxint** meaning that it occupies the entire scrollbar. In practice, there is a practical limit to how small the slider can be. If the slider is set too small, it will be set to the minimum size, and no error will occur. If the size set is negative, then an error will result.

If the window file **f** does not exist, the "output" file will be used by default.

The size of the scrollbar is set to a reasonable default if it is never specifically set. This is typically a fairly small size that is still easy to press and manipulate by the user. This allows the scrollbar to be used when slider sizing is not supported by the program.

The meaning of the scrollbar slider size is up to the program. However, it is typically used to indicate how much of the data is onscreen. For example, if a document has %50 of its content currently displayed, then the slider would be set to %50 of the scrollbar.

Exceptions: **WidgetNotFound, InvalidScrollBarSize**

procedure numselboxsiz[g]([var f: text;] l, u: integer; var w, h: integer);

Finds the width and height of a number select box for window **f**, with lower number limit **l** and upper number limit **u**. The width required is returned in **w**, and the height in **h**. The minimum width and height is determined by the maximum length of the number to be displayed, with borders and up/down arrows considered. This can be used without adding extra space.

Exceptions: None

procedure numselbox[g]([var f: text;] x1, y1, x2, y2: integer; l, u: integer; id: integer);

Creates a number select box for window **f**, in the rectangle **x1**, **y1**, **x2**, **y2**, with lower number limit **l**, and upper number limit **u**. The default number appearing in the box is set to the lower limit. The number select box allows the user to edit the number, or use up/down arrow controls to select the number. Any digits typed into the edit section are limited to the digits 0-9, and negative numbers are not allowed. When the user presses enter to the number edit box, an event, **etnumbox** will be sent, which includes the number selected.

Exceptions: **WidgetIdDuplicate**

procedure editboxsiz[g]([var f: text;] view s: string; var w, h: integer);

Finds the size of an edit box for window **f**, with face text string **s**. The width is returned in **w**, and the height in **h**. The string passed is a dummy, and will not be used for any purpose other than as a reference to determine the width of the required edit box. The string should contain text that is representative of the string to be edited. This could be the string that you plan to place in the edit box as its default, or it could be the worst case contents of the edit box. For example, if 8 characters is the planned edit width, the string "WWWWWWWW" (8 "W" characters) would be the largest width of edit possible. The edit box is sized to be the minimum appropriate, and can be used without extra added space.

Exceptions: None

procedure editbox[g]([var f: text;] x1, y1, x2, y2: integer; id: integer);

Creates an edit box for window **f**, in rectangle **x1**, **y1**, **x2**, **y2**, with logical identifier **id**. Edit boxes can be used to allow the user to enter any text. The text within an edit box can be set by **putwidgettext**, and retrieved by **getwidgettext**. This can occur at any time. When the user presses enter in the edit box, it sends an **etedtbox** event. The program can then retrieve the text from the edit box.

Exceptions: **WidgetIdDuplicate**

procedure progbarsiz[g]([var f: text;] var w, h: integer);

Finds the size of a progress bar for window **f**. The width is returned in **w**, and the height in **h**. For systems that can size progress bars arbitrarily, the height is returned as the size that matches others used in the system. The width is a suggestion, and can be ignored.

Exceptions: None

procedure progbar[g]([var f: text;] x1, y1, x2, y2: integer; id: integer);

Creates a progress bar in window **f**, in rectangle **x1**, **y1**, **x2**, **y2**, with logical identifier **id**. The progress bar starts by default at 0, and is entirely operated by the program with **progbarpos** calls.

Exceptions: **WidgetIdDuplicate**

procedure progbarpos([var f: text;] id: integer; pos: integer);

Sets the progress bar in window **f**, with logical identifier **id**, to the position **pos**. The position is a ratioed **maxint** number, from 0 to **maxint**. 0 indicates "no progress", and **maxint** indicates "complete". Because of rounding, it is recommended that the program specifically set **maxint** at completion, instead of using a formula.

Exceptions: **WidgetNotFound, InvalidProgressBarPosition**

procedure listboxsiz[g]([var f: text;] sp: strptr; var w, h: integer);

Finds the required size of a **listbox** for window **f**, with string list **sp**. The required width is returned in **w**, and the required height is returned in **h**. A listbox is sized such that all of the strings in the string list can be presented in it, with borders added. No extra space is required.

Exceptions: None

procedure listbox[g]([var f: text;] x1, y1, x2, y2: integer; sp: strptr; id: integer);

Creates a **listbox** for window **f**, in rectangle **x1**, **y1**, **x2**, **y2**, with string list **sp**, and logical identifier **id**. A **listbox** contains a series of strings that can be selected by the user. When the user clicks a string, the event **etlstbox** will be sent, which contains the number of the selected string in list order. For example, the first string in the list would be 1, and second string in the list 2, etc. If there is not enough room in the height of the **listbox** for all strings in the list to be presented, then the widget will use a compression method to fit the available space. This is typically done by allowing the user to scroll through the selections. If there is not enough width for the strings in the list, they are typically clipped at the right.

Exceptions: **WidgetIdDuplicate, NoStringSpace**

procedure dropboxsiz[g](var f: text; sp: strptr; var cw, ch, ow, oh: integer);

Finds the size of a drop box for window **f**, with string list sp. The closed width is returned in **cw**, and the closed height in **h**. The open width is returned in **ow**, and the height in **oh**. Drop boxes are used to display a list of selections as in a **listbox**, but they occupy less space than a **listbox**. Dropboxes have two bounding rectangles, one is its dimensions when closed, and another when the user drops it down, or opens it. Both sizes are returned. This allows layout planning for both modes of the widget. Generally, the closed size is used to plan placement, then the open size is used to check if the widget will extend past the edges of the window when open.

Exceptions: None

procedure dropbox[g]([var f: text;] x1, y1, x2, y2: integer; sp: strptr; id: integer);

Creates a **dropbox** in the window **f**, within rectangle **x1**, **y1**, **x2**, **y2**, for string list **sp**, with logical identifier **id**. The bounding rectangle for a drop box specifies its open mode, where the user has selected and dropped down the list of items. If the size specified is greater than or equal to the open size, as determined by **dropboxsiz**, then the entire **dropbox** will be presented. If the size is between the closed size and the open size, the system will attempt to work around the fact that the entire list cannot be dropped down. This is typically done by allowing the user to scroll though the list. If the size is less than the closed size, the **dropbox** will be clipped.

When a string within the drop box is selected, it will send an **etdrpbox** event. It contains the sequential number of the string that was selected. For example, the first string sends 1, the second in the list sends 2, etc.

Exceptions: **WidgetIdDuplicate, NoStringSpace**

procedure dropeditboxsiz[g]([var f: text;] sp: strptr; var cw, ch, ow, oh: integer);

Finds the size of a drop edit box for window **f**, with string list sp. The closed width is returned in **cw**, and the closed height in **h**. The open width is returned in **ow**, and the height in **oh**. Drop edit boxes are used to display a list of selections, and acts as a combination of a list and edit box, but they occupy less space than a **listbox**. Drop edit boxes have two bounding rectangles, one is its dimensions when closed, and another when the user drops it down, or opens it. Both sizes are returned. This allows layout planning for both modes of the widget. Generally, the closed size is used to plan placement, then the open size is used to check if the widget will extend past the edges of the window when open.

Exceptions: None

procedure dropeditbox[g]([var f: text;] x1, y1, x2, y2: integer; sp: strptr; id: integer);

Creates a drop edit box in the window **f**, within rectangle **x1**, **y1**, **x2**, **y2**, for string list **sp**, with logical identifier **id**. The bounding rectangle for a drop edit box specifies its open mode, where the user has selected and dropped down the list of items. If the size specified is greater than or equal to the open size, as determined by **dropboxsiz**, then the entire **dropbox** will be presented. If the size is between the closed size and the open size, the system will attempt to work around the fact that the entire list cannot be dropped down. This is typically done by allowing the user to scroll though the list. If the size is less than the closed size, the **dropbox** will be clipped.

When a drop box string is selected, or enter is hit while editing, it sends the event **etdrebox**. There is no other information associated with this event. Since the text is editable, it could be anything, and may not match one of the list entries. Instead, the program should use **getwidgettext** to retrieve the result of the edit.

Drop edit boxes default to a blank edit string. The idea of the drop edit box is that the user can simply use it as an edit box to enter the needed data, or drop down a list of preselected items. If you wish to make one of the string list items the default, or even a text that is not on the list, use the **putwidgettext** to initialize the edit field.

Exceptions: **WidgetIdDuplicate, NoStringSpace**

procedure slidehorizsiz[g]([var f: text;] var w, h: integer);

Finds the size of a horizontal scrollbar for window **f**. The required width is returned in **w**, and the required height in **h**. The height of a slider is chosen so that they match other slidebars used in the system. The width is a suggestion, and can be ignored.

Exceptions: None

procedure slidehoriz[g]([var f: text;] x1, y1, x2, y2: integer; mark: integer; id: integer);

Creates a horizontal slider in window **f**, in bounding rectangle **x1**, **y1**, **x2**, **y2**, with **mark** number of tick marks, and a logical identifier **id**. Sliders give a convenient way to select from a range of values. The system will either size the slider to fit the given rectangle, or choose the slider representation that is as large as possible, but still fits the given rectangle. It is not guaranteed that the slider will fill the entire rectangle. This means that it is important for the entire background under the rectangle to be set to the standard background color.

When the slider is moved by the user, it generates a **etsldpos** event. This event carries the new position of the slider, which is a ratioed **maxint** number, from 0 to **maxint**. 0 means the slider is at the extreme left, and **maxint** means the slider is at the extreme right.

There is no guarantee as to when a slider generates its events. It can generate them as the slider is moved, or it may wait until the user moves, and then releases the slider to generate events. Sliders automatically update the position of the slider, and do not need to be set by the program.

The number of tick marks given by **mark** are evenly distributed across the slider. If **mark** is zero, then no tick marks are placed at all. Tick marks have no other effect besides appearing on the slider.

Exceptions: **WidgetIdDuplicate**

procedure slidevertsiz[g]([var f: text;] var w, h: integer);

Finds the size of a vertical slider for window **f**. The required width is returned in **w**, and the required height in **h**. The width of a slider is chosen so that they match other slidebars used in the system. The height is a suggestion, and can be ignored.

Exceptions: None

procedure slidevert[g]([var f: text;] x1, y1, x2, y2: integer; mark: integer; id: integer);

Creates a vertical slider in window **f**, in bounding rectangle **x1**, **y1**, **x2**, **y2**, with mark number of tick marks, and a logical identifier **id**. Sliders give a convenient way to select from a range of values. The system will either size the slider to fit the given rectangle, or choose the slider representation that is as large as possible, but still fits the given rectangle. It is not guaranteed that the slider will fill the entire rectangle. This means that it is important for the entire background under the rectangle to be set to the standard background color.

When the slider is moved by the user, it generates a **etsldpos** event. This event carries the new position of the slider, which is a ratioed **maxint** number, from 0 to **maxint**. 0 means the slider is at the extreme top, and **maxint** means the slider is at the extreme bottom.

There is no guarantee as to when a slider generates its events. It can generate them as the slider is moved, or it may wait until the user moves, and then releases the slider to generate events. Sliders automatically update the position of the slider, and do not need to be set by the program.

The number of tick marks given by **mark** are evenly distributed across the slider. If **mark** is zero, then no tick marks are placed at all. Tick marks have no other effect besides appearing on the slider.

Exceptions: **WidgetIdDuplicate**

procedure tabbarsiz[g[([var f: text;] tat, tar, tab, tal: boolean; cw, ch: integer; var w, h, ox, oy: integer);

Finds the size of a tabbar, in window **f**, with tab side active state **tat**, **tar**, **tab**, **tal**, client width **w**, and client height **h**. The required width is returned in **cw**, the height in **ch**, and the client offset in **ox** and **oy**. The size of a tabbar is enough to hold the height of the labeling font (in whatever orientation), plus border areas, any selection scrolling arrows, and the client area.

A tabbar can have tabs on any of its four sides. The enables for the tabs on a side are:

| | |
|---|---|
| **tat** | Top |
| **tar** | Right |
| **tab** | Bottom |
| **tal** | Left |

They are in clockwise order starting from the top.

Exceptions: None

procedure tabbarclient[g[([var f: text;] tat, tar, tab, tal: boolean; w, h: integer; var cw, ch, ox, oy: integer);

Finds the size of a tabbar client area, in window **f**, with tab side active state **tat**, **tar**, **tab**, **tal**, tabbar width **w**, and tab bar height **h**. The client width is returned in **cw**, the height in **ch**, and the client offset in **ox** and **oy**. This procedure is used to find the client area for a specific size of tabbar.

A tabbar can have tabs on any of its four sides. The enables for the tabs on a side are:

| | |
|---|---|
| **tat** | Top |
| **tar** | Right |
| **tab** | Bottom |
| **tal** | Left |

They are in clockwise order starting from the top.

Exceptions: None

procedure tabbar[g]([var f: text;] x1, y1, x2, y2: integer; spt, spr, spb, spl: strptr; id: integer);

Creates a tab bar, in the window **f**, with string lists **spt, spr, spb, spl**, and logical identifier id. A tabbar gives the user a paradigm of a book with tabs on the side. The list of tabs, which are specified by the program, each generate events. The program establishes the view to be selected in the client area at the center of the tabbar, then it uses the tab events to switch the views in the client. If there is not enough area to display the full list of tabs, the system will allow the user to scroll through them with arrows.

To locate the tabbar and client, the **tabbarsiz** call is used to establish the size and client offset. Then, the client is offset into the tabbar. The client widgets or child windows must be created after the tabbar in order to be placed to the front of the stacking order.

When a tab is selected, it generates an **ettabbar** event, which contains both the logical tabbar id, and the number of the string list that was selected. This number will be the number in the string list. For example, the first string in the list will be 1, the second 2, etc.

A tabbar can have tabs on any of its four sides. The string lists that form the tabs on a side are:

| | |
|---|---|
| **spt** | Top |
| **spr** | Right |
| **spb** | Bottom |
| **spl** | Left |

They are in clockwise order starting from the top.

If tabs are not required on a given side, the string list for that side is passed as nil. When a tab list is not active on a side, the client area will be extended to cover the tabs on that side. If no tab string list is active, then a tabbar is equivalent to a group.

Exceptions: **WidgetIdDuplicate, UnableCreateTab**

procedure tabsel([var f: text;] id, tn: integer);

Select tab in tab bar. Causes the tab **tn** in the tab bar **id** in the window **f**, to enter the selected state. **tn** is the number of the string list item to select. For example, the first string in the list will be 1, the second 2, etc.

Exceptions: **WidgetNotFound, InvalidTabSelect**

procedure alert(view title, msg: string);

Creates an alert dialog, with window title **title**, and client message **msg**. The alert dialog is a freestanding window that is placed at the front of the desktop stacking order. It is generally used to display an error, warning or other attention condition. The window title should tell the user what application is generating the alert, and the client message should give the error or warning. The user dismisses the dialog, and the program holds until the dialog completes.

Exceptions: None

procedure querycolor(var r, g, b: integer);

Creates a color select dialog. The dialog "flows through" to set its parameters. When called, **r** contains the default red, **g** the default green, and **b** the default blue colors. These defaults are used to set the dialog default selection. When the user chooses a color, the same parameters return the new selection. If the user cancels, or leaves the default selection alone, the input colors will simply be left as the default output colors. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

Exceptions: None

procedure queryopen(var s: pstring);

Creates an open file dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default filename to open (which could be null). This default is used to initialize the dialog default. When the user edits a filename, that is then returned in **s** as well. The input and output strings are not the same even if the user chooses the default. The result string must be disposed of by the caller, and the default string supplied to the dialog is allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

If the user cancels, a null string is returned (a string with zero length, not a nil pointer).

Exceptions: **UnableCreateFileDialog**

procedure querysave(var s: pstring);

Creates an save file dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default filename to save (which could be null). This default is used to initialize the dialog default. When the user edits a filename, that is then returned in **s** as well. The input and output strings are not the same even if the user chooses the default. The result string must be disposed of by the caller, and the default string supplied to the dialog is allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

If the user cancels, a null string is returned (a string with zero length, not a nil pointer).

Exceptions: **UnableCreateFileDialog**

procedure queryfind(var s: pstring; var opt: qfnopts);

Creates a find string dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default search string (which could be null). This default is used to initialize the dialog default. When the user edits a search, that is then returned in **s** as well. The input and output strings are not the same even if the user chooses the default. The result string must be disposed of by the caller, and the default string supplied to the dialog is allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

The find dialog may set one or more of several options from the set provided in **opt**. These are set before the call, and they are used to initialize the defaults in the dialog. When the dialog terminates, the new state of the options are returned in **opt** as well. If the dialog does not implement a particular option, then the input value will simply be copied to the output value without change.

If the user cancels, a null string is returned (a string with zero length, not a nil pointer).

Exceptions: **UnableCreateFindDialog**

procedure queryfindrep(var s, r: pstring; var opt: qfropts);

Creates a find/replace string dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default search string (which could be null), and **r** contains the default replacement string (which could be null). This default is used to initialize the dialog default. When the user edits a search, that is then returned in **s** and **r** as well. The input and output strings are not the same even if the user chooses the default. The result strings must be disposed of by the caller, and the default strings supplied to the dialog are allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

The find dialog may set one or more of several options from the set provided in **opt**. These are set before the call, and they are used to initialize the defaults in the dialog. When the dialog terminates, the new state of the options are returned in **opt** as well. If the dialog does not implement a particular option, then the input value will simply be copied to the output value without change.

If the user cancels, a null string is returned (a string with zero length, not a nil pointer).

Exceptions: **UnableCreateFindDialog**

procedure queryfont([var f: text;] var fc, s, fr, fg, fb, br, bg, bb: integer; var effect: qfteffects);
procedure queryfont([var f: text;] var fc, s: integer; var fcl, bcl: color; var effect: qfteffects);

Creates a query font dialog. The dialog "flows through" to set its parameters. When called, **fc** contains the font code, **s** contains the size, **fr**, **fg** and **fb** contain the foreground red, green and blue colors, **br**, **bg**, **bb** contains the background red, green and blue colors, and effect contains a set of font effects. These values are used to initialize the dialog defaults. The user then sets any or all of the parameters, and the results are copied back to the same output parameters. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

If the dialog does not have a particular feature such as color set ability or one or more effects, the input for that parameter is simply copied to the output.

Exceptions: **UnableCreateFontDialog**

## N.17 Events and Callbacks In widgets

For each item, both the event record section and the virtual procedure is presented. See the description of the event record (L.21 “Declarations”) for the format of the entire record.

Event: etbutton
**virtual procedure** evbutton(id: integer);

The button with identifier **[but]id** was pressed.

Event: etchkbox
**virtual procedure** evchkbox(id: integer);

The checkbox with identifier **[chbx]id** was selected.

Event: etradbut
**virtual procedure** evradbut(id: integer);

The radio button with identifier **id** was selected.

Event: etsclull
**virtual procedure** evsclull(id: integer);

The scrollbar with identifier **id** had its up or left line button pressed.

Event: etscldrl
**virtual procedure** evscldrl(id: integer);

The scrollbar with identifier **id** had its down or right line button pressed

Event: etsclulp
**virtual procedure** evsclulp(id: integer);

The scrollbar with identifier **id** had its up or left page button pressed.

Event: etscldrp
**virtual procedure** evscldrp(id: integer);

The scrollbar with identifier **id** had its down or right page button pressed.

Event: etsclpos
**virtual procedure** evsclpos(id: integer; pos: integer);

The scrollbar with identifier **id** was repositioned to pos. The value **pos** is maxint ratio'ed, with 0 indicating top or left, and maxint indicating bottom or right.

Event: etedtbox
**virtual procedure** evedtbox(id: integer);

The editbox with identifier **id** was given an enter key. This means that the text in the editbox is complete, and can be retrieved by **getwidgettext**.

Event: etnumbox
**virtual procedure** evnumbox(id: integer; sl: integer);

The numselbox with the identifier **id** was entered with the number **sl**. **sl** directly corresponds to the number selected.

Event: etlstbox
**virtual procedure** evlstbox(id: integer; sl: integer);

The listbox with the identifier **id** was entered with the logical select **sl**. The logical select **sl** gives the number of the string selected in the order used to create the listbox, with 1 indicating the first string, 2 the second, etc.

Event: etdrpbox
**virtual procedure** evdrpbox(id: integer; sl: integer);

The listbox with the identifier **id** was entered with the logical select **sl**. The logical select **sl** gives the number of the string selected in the order used to create the listbox, with 1 indicating the first string, 2 the second, etc.

Event: etdrebox
**virtual procedure** evdrebox(id: integer);

The dropeditbox with identifier **id** was given an enter key. This means that the text in the editbox is complete, and can be retrived by **getwidgettext**.

Event: etsldpos
**virtual procedure** evsldpos(id: integer; pos: integer);

The slider with identifier **id** was repositioned to pos. The value **pos** is maxint ratio'ed, with 0 indicating top or left, and maxint indicating bottom or right.

Event: ettabbar
**virtual procedure** evtabbar(id: integer; tor: tabori; sel: integer);

The tabbar with the identifier id had a tab selected with the orientation tor and the string number sel. tor indicates which in which string list the select occurred, top, bottom, left, right. **sel** indicates which string in that list was selected, with 1 being the first, 2 being the second, etc.

# O Annex: Sound Library

**sound** adds both a synthesizer interface via the MIDI standard, and the ability to play wave files. It implements or takes advantage of a sequencer that programs the exact time at which each of the events to make a complex combination of sounds occur.

The MIDI interface is defined as a serial interface, but the target of a MIDI port can be a standard serial MIDI daisy chain, another type of interface that carries MIDI commands (such as USB), or simply terminate in an internal sound card or even a software synthesizer.

## O.1 Ports

A port is the basic MIDI output device. Typically, a computer has two of them, the sound card internal to the computer, and the external MIDI jack. These ports are labeled **synth_out** and **synth_ext**, respectively. A synthesizer output is opened with **opensynthout**. It can be closed with **closesynthout**. All synthesizer ports are automatically closed when the program closes.

```
synth_out = 1; { the default output synth for host }
synth_ext = 2; { the default output to external synth }
```

The total number of synthesizer output ports is found by **synthout**. Notes

```
type note = 1..128; { note number for midi }
```

The basic work of making music is playing notes. MIDI can play 128 notes, numbered from 1 to 128. This ranges in frequency from 8 Hertz, or cycles per second, to 12 Kilohertz. This is approximately the range of human hearing. MIDI can also change each note in frequency enough to move it to the note next to it (and then some), so MIDI is able to reach any frequency desired.

Humans perceive a frequency that is 4 times higher as being only twice as high. If a musical note is doubled in frequency, it will be perceived as the same note one octave higher. There are twelve notes in an octave. In the lowest octave, they are:

```
{ the notes in the lowest octave }

note_c        = 1;
note_c_sharp  = 2;
note_d_flat   = 2;
note_d        = 3;
note_d_sharp  = 4;
note_e_flat   = 4;
note_e        = 5;
note_f      = 6;
note_f_sharp  = 7;
note_g_flat   = 7;
note_g        = 8;
note_g_sharp  = 9;
note_a_flat   = 9;
note_a        = 10;
note_a_sharp  = 11;
note_b_flat   = 11;
note_b        = 12;

The bases of the octaves are:

{ the octaves of midi, add to note to place in that octave }

octave_1  = 0;
octave_2  = 12;
octave_3  = 24;
octave_4  = 36;
octave_5  = 48;
octave_6  = 60;
octave_7  = 72;
octave_8  = 84;
octave_9  = 96;
octave_10 = 108;
octave_11 = 120;
```

So any note in any octave can be found by:

note+octave

For example, C in the 6th Octave:

note_c+octave_6

Notes are activated in MIDI by the **noteon** procedure, and deactivated by **noteoff**. Each of these calls may take:

- A Port
- A time
- A channel
- A note
- A volume

Each of these parameters will be presented separately. The time to play will be discussed below. For now, it can be zero, which means "play it now". The channel is the instrument type to play it to, for instance, a piano, or an organ, or a tuba. The note is the logical note number we saw above, one of the 128 MIDI notes. The volume gives the volume the particular note is to be played at. A piano note can be louder if hit harder.

A note can either last forever, until turned off with **noteoff**, or it can stop on its own. For example, an organ plays as long as you hold the key down, but a string instrument plays a note when the string is plucked, then dies away. **noteoff** need not be used for these instruments, but can still be used to cause the note to be "clipped" off early, much as if the player put a hand on the string to stop it. Similarly, a **noteon** can be used to restart the note, even while it is playing.

## O.2 Channels and Instruments

```
type channel    = 1..16;  { channel number }
     instrument = 1..128; { instrument number }
```

MIDI has from 1 to 16 logical channels, indexed by a logical channel number. Although there are 128 instruments, only one can be played at any one time. To play an instrument, it must be assigned to a channel. This is done with **instchange.** The instruments available are:

```
{ Standard GM instruments }

{ Piano }

inst_acoustic_grand         = 1;
inst_bright_acoustic        = 2;
inst_electric_grand         = 3;
inst_honky_tonk             = 4;
inst_electric_piano_1       = 5;
inst_electric_piano_2       = 6;
inst_harpsichord            = 7;
inst_clavinet               = 8;
```

```
{ Chromatic percussion }

inst_celesta                = 9;
inst_glockenspiel           = 10;
inst_music_box              = 11;
inst_vibraphone             = 12;
inst_marimba                = 13;
inst_xylophone              = 14;
inst_tubular_bells          = 15;
inst_dulcimer               = 16;

{ Organ }

inst_drawbar_organ          = 17;
inst_percussive_organ       = 18;
inst_rock_organ             = 19;
inst_church_organ           = 20;
inst_reed_organ             = 21;
inst_accoridan              = 22;
inst_harmonica              = 23;
inst_tango_accordian        = 24;

{ Guitar }

inst_nylon_string_guitar    = 25;
inst_steel_string_guitar    = 26;
inst_electric_jazz_guitar   = 27;
inst_electric_clean_guitar  = 28;
inst_electric_muted_guitar  = 29;
inst_overdriven_guitar      = 30;
inst_distortion_guitar      = 31;
inst_guitar_harmonics       = 32;

{ Bass }

inst_acoustic_bass          = 33;
inst_electric_bass_finger   = 34;
inst_electric_bass_pick     = 35;
inst_fretless_bass          = 36;
inst_slap_bass_1            = 37;
inst_slap_bass_2            = 38;
inst_synth_bass_1           = 39;
inst_synth_bass_2           = 40;
```

```
{ Solo strings }

inst_violin                 = 41;
inst_viola                  = 42;
inst_cello                  = 43;
inst_contrabass             = 44;
inst_tremolo_strings        = 45;
inst_pizzicato_strings      = 46;
inst_orchestral_strings     = 47;
inst_timpani                = 48;

{ Ensemble }

inst_string_ensemble_1     = 49;
inst_string_ensemble_2     = 50;
inst_synthstrings_1        = 51;
inst_synthstrings_2        = 52;
inst_choir_aahs            = 53;
inst_voice_oohs            = 54;
inst_synth_voice           = 55;
inst_orchestra_hit         = 56;

{ Brass }

inst_trumpet               = 57;
inst_trombone              = 58;
inst_tuba                  = 59;
inst_muted_trumpet         = 60;
inst_french_horn           = 61;
inst_brass_section         = 62;
inst_synthbrass_1          = 63;
inst_synthbrass_2          = 64;

{ Reed }

inst_soprano_sax           = 65;
inst_alto_sax              = 66;
inst_tenor_sax             = 67;
inst_baritone_sax          = 68;
inst_oboe                  = 69;
inst_english_horn          = 70;
inst_bassoon               = 71;
inst_clarinet              = 72;
```

```
{ Pipe }

inst_piccolo               = 73;
inst_flute                 = 74;
inst_recorder              = 75;
inst_pan_flute             = 76;
inst_blown_bottle          = 77;
inst_skakuhachi            = 78;
inst_whistle               = 79;
inst_ocarina               = 80;

{ Synth lead }

inst_lead_1_square          = 81;
inst_lead_2_sawtooth        = 82;
inst_lead_3_calliope        = 83;
inst_lead_4_chiff           = 84;
inst_lead_5_charang         = 85;
inst_lead_6_voice           = 86;
inst_lead_7_fifths          = 87;
inst_lead_8_bass_lead       = 88;

{ Synth pad }

inst_pad_1_new_age          = 89;
inst_pad_2_warm             = 90;
inst_pad_3_polysynth        = 91;
inst_pad_4_choir            = 92;
inst_pad_5_bowed            = 93;
inst_pad_6_metallic         = 94;
inst_pad_7_halo             = 95;
inst_pad_8_sweep            = 96;

{ Synth effects }

inst_fx_1_rain              = 97;
inst_fx_2_soundtrack        = 98;
inst_fx_3_crystal           = 99;
inst_fx_4_atmosphere        = 100;
inst_fx_5_brightness        = 101;
inst_fx_6_goblins           = 102;
inst_fx_7_echoes            = 103;
inst_fx_8_sci_fi            = 104;
```

```
{ Ethnic }

inst_sitar                  = 105;
inst_banjo                  = 106;
inst_shamisen               = 107;
inst_koto                   = 108;
inst_kalimba                = 109;
inst_bagpipe                = 110;
inst_fiddle                 = 111;
inst_shanai                 = 112;

{ Percussive }

inst_tinkle_bell            = 113;
inst_agogo                  = 114;
inst_steel_drums            = 115;
inst_woodblock              = 116;
inst_taiko_drum             = 117;
inst_melodic_tom            = 118;
inst_synth_drum             = 119;
inst_reverse_cymbal         = 120;

{ Sound effects }

inst_guitar_fret_noise     = 121;
inst_breath_noise          = 122;
inst_seashore              = 123;
inst_bird_tweet            = 124;
inst_telephone_ring        = 125;
inst_helicopter            = 126;
inst_applause              = 127;
inst_gunshot               = 128;
```

When MIDI starts up, all channels are assigned logical instrument number 1, an acoustical grand piano, with the exception of channel 10.

The MIDI channel system allows an “arrangement” to be created from different instruments. Each channel is configured with an instrument, then used to play a sequence. The computer can quickly change instruments during the music, and start an entirely different kind of music without skipping a beat. It is good practice not to count on an instrument being able to complete a note that is playing if it is swapped out of its channel for another instrument.

Channel 10 is an exception. This channel is always reserved for percussion (or drum) sounds. In this channel, the notes sent have a special meaning. In fact, each note selects a different instrument:

```
chan_drum = 10; { the GM drum channel }

{ Drum sounds, activated as notes to drum instruments }

note_acoustic_bass_drum   = 35;
note_bass_drum_1          = 36;
note_side_stick           = 37;
note_acoustic_snare       = 38;
note_hand_clap            = 39;
note_electric_snare       = 40;
note_low_floor_tom        = 41;
note_closed_hi_hat        = 42;
note_high_floor_tom       = 43;
note_pedal_hi_hat         = 44;
note_low_tom              = 45;
note_open_hi_hat          = 46;
note_low_mid_tom          = 47;
note_hi_mid_tom           = 48;
note_crash_cymbal_1       = 49;
note_high_tom             = 50;
note_ride_cymbal_1        = 51;
note_chinese_cymbal       = 52;
note_ride_bell            = 53;
note_tambourine           = 54;
note_splash_cymbal        = 55;
note_cowbell              = 56;
note_crash_cymbal_2       = 57;
note_vibraslap            = 58;
note_ride_cymbal_2        = 59;
note_hi_bongo             = 60;
note_low_bongo            = 61;
note_mute_hi_conga        = 62;
note_open_hi_conga        = 63;
note_low_conga            = 64;
note_high_timbale         = 65;
note_low_timbale          = 66;
note_high_agogo           = 67;
note_low_agogo            = 68;
note_cabasa               = 69;
note_maracas              = 70;
note_short_whistle        = 71;
note_long_whistle         = 72;
note_short_guiro          = 73;
note_long_guiro           = 74;
note_claves               = 75;
note_hi_wood_block        = 76;
note_low_wood_block       = 77;
note_mute_cuica           = 78;
note_open_cuica           = 79;
```

```
note_mute_triangle        = 80;
note_open_triangle        = 81;
```

Percussion instruments always stop themselves.

The same instrument can be assigned to multiple channels. This allows an instrument harmonize with itself, playing overlapping notes.

## O.3 Volume

The volume can be set for each individual note. It can also be set for the entire synthesizer port by **volsynth**. Volume can even be set for each channel by **volsynthchan**.

The volume is "**maxint** ratioed". It exists as a value from 0 to **maxint**, where 0 off (no volume) and **maxint** is full on. It is not decibel compensated, meaning that **maxint** div 2 is not half volume.

Balance between left and right can be set for each channel with **balance**. It's still **maxint** ratioed, except that 0 means middle, **maxint** means full right, and -**maxint** means full left.

## O.4 Time and the Sequencer

MIDI does not have a concept of time built into the protocol. All notes or events sent to the MIDI port are assumed to happen "Now".

**sound** has sequencer support that is used by setting a time on each event call. If the time is 0, it means to send the note or event to the MIDI port immediately, otherwise the sequencer schedules the event to occur at the indicated time.

To start **sound**'s sequencer, the **starttime** is used, which starts a 100us counter running (it ticks every 10,000th of a second). Then, each time is specified relative to that running timer. The current time on the sequencer can be found with **curtime**, so the required time can be specified as an offset from that:

curtime+10000

means a time that is one second in the future.

This example dumps a "fanfare" into the MIDI port using the sequencer. It will be played using the time specified in the **noteon** and **noteoff** calls.

```
program p;

uses sound;

const second = 10000;           { one second }
      osec   = second div 8;    { 1/8 second }

begin

   starttime; { start sequencer }

   noteon(synth_out,  0,              1, note_c+octave_6, maxint);
   noteoff(synth_out, curtime+osec*2,  1, note_c+octave_6, maxint);
   noteon(synth_out,  curtime+osec*3,  1, note_d+octave_6, maxint);
   noteoff(synth_out, curtime+osec*4,  1, note_d+octave_6, maxint);
   noteon(synth_out,  curtime+osec*5,  1, note_e+octave_6, maxint);
   noteoff(synth_out, curtime+osec*6,  1, note_e+octave_6, maxint);
   noteon(synth_out,  curtime+osec*7,  1, note_f+octave_6, maxint);
   noteoff(synth_out, curtime+osec*8,  1, note_f+octave_6, maxint);
   noteon(synth_out,  curtime+osec*9,  1, note_e+octave_6, maxint);
   noteoff(synth_out, curtime+osec*10, 1, note_e+octave_6, maxint);
   noteon(synth_out,  curtime+osec*11, 1, note_d+octave_6, maxint);
   noteoff(synth_out, curtime+osec*13, 1, note_d+octave_6, maxint);

end.
```

The fanfare plays, and the program goes on to other work, or waits for the sequenced time to pass by setting a timer to the time required to finish it.

If a note is output (or other action) with a 0 time while the sequencer is running, it will still occur immediately. Time 0 always means "now". The sequencer is a "flow through" model. Actions and notes can be timed with the sequencer, or by the program, or any combination thereof.

When the sequencer is no longer required, **stoptime** stops it. Doing that can save processor time, and probably free up system timers.

## O.5 Effects

There are many effects in MIDI that can be applied to output notes. However, there is no requirement for the system to implement them. Few of the effects are implemented on most computer sound cards or software synthesizers.

**attack** adjusts the "attack time" of each note.

**release** adjusts the release or "decay" time of the note.

**reverb** sets the amount of reverberation, or a series of repetitions of the note with delay.

**vibrato** sets the vibrato, which is a pulsating pitch change.

**chorus** sets the chorus effect, which is an echo of the same note with a delay and possible pitch change..

**phaser** sets the phaser effect, which is a series of peaks and valleys in the frequency spectrum of the note.

**brightness** sets the brightness, or VCF cutoff frequency.

**timbre** [need a definition of timbre].

**aftertouch** sets the amount of time or pressure used to sustain a key pressed.

**pressure** sets the amount of pressure applied to a key.

**legato** sets the note to be played shorter than normal.

**portamento** sets the note to "slide" or smoothly change into the next note.

Some missing effects can be simulated by other means. As an example, **release** control can be emulated by putting the instrument to control in its own channel, sounding the note, then lowering the volume in steps until 0, then turning the note off.

## O.6 Pitch Changes

If a frequency is needed that is not exactly on a note, it can be "bent" with **pitch**. The pitch change is none for 0, and by default, one note up or down. In other words, the default pitch change range is one note up or down. A D note can be bent downwards to C, or upwards to E. The term "bend" comes from bending a string to change the note.

The default range of pitch changes can also be changed, by **pitchrange**. The range is a ratioed 0..**maxint**. 0 means no pitch range at all (disabled), and **maxint** means the full 128 notes worth of pitch range. What you pick up with total range, you lose in fine control. If the pitch range is **maxint**, each step of pitch change is going to be very coarse.

## O.7 Prerecorded MIDI

We don't have to make all our MIDI commands on the fly. In fact, we can forget doing any MIDI, and just play back prerecorded MIDI files with **playsynth**. The format is system defined. Note that even though the prerecorded MIDI file has its own timing, it is played relative to the clock start position that is indicated for it.

## O.8 Waveform Files

Waveform files are how we get arbitrary sounds into the computer. Anything, for any length, can be played via the waveform files. Waveform outputs go out via their own ports separate from MIDI ports. Like MIDI, however, they are selected via logical numbers from 1 to n, where n is the maximum number of waveform output devices on the computer. The total number of waveform devices present in the system can be found via **waveout**. By convention, the normal wave output device is 1.

Waveform devices must be opened and closed individually. They are opened with **openwaveout**, and closed with **closewaveout**.

A waveform file is played with **playwave**. Waveform files are usually very system dependent, so the exact format of the file will be different for different systems.

As with MIDI files, waveforms have their own timing, and are simply played relative to the indicated start time.

The playback volume for waveform files is adjusted separately from MIDI with the **volwave**.

It is possible that the implementation will only be able to play one waveform file at a time. In this case, the behavior will be to stop any currently playing waveform file and start the new one, if a new waveform play is ordered before the previous one has finished. High quality implementation will be capable of playing multiple waveform files at once, typically via mixing of the files.

## O.9 Synthesizer objects

The functionality of a synthesizer is available in the form of a class:

```
module sound;

class synth(int port);

procedure noteon(t: integer; c: channel; n: note; v: integer);
procedure noteoff(t: integer; c: channel; n: note; v: integer);
procedure instchange(t: integer; c: channel; i: instrument);
procedure attack(t: integer; c: channel; at: integer);
procedure release(t: integer; c: channel; rt: integer);
procedure legato(t: integer; c: channel; b: boolean);
procedure portamento(t: integer; c: channel; b: boolean);
procedure vibrato(t: integer; c: channel; v: integer);
procedure volsynthchan(t: integer; c: channel; v: integer);
procedure porttime(t: integer; c: channel; v: integer);
procedure balance(t: integer; c: channel; b: integer);
procedure pan(t: integer; c: channel; b: integer);
procedure timbre(t: integer; c: channel; tb: integer);
procedure brightness(t: integer; c: channel; b: integer);
procedure reverb(t: integer; c: channel; r: integer);
procedure tremulo(t: integer; c: channel; tr: integer);
procedure chorus(t: integer; c: channel; cr: integer);
procedure celeste(t: integer; c: channel; ce: integer);
procedure phaser(t: integer; c: channel; ph: integer);
procedure aftertouch(t: integer; c: channel; n: note; at: integer);
procedure pressure(t: integer; c: channel; n: note; pr: integer);
procedure pitch(t: integer; c: channel; pt: integer);
procedure pitchrange(t: integer; c: channel; v: integer);
procedure mono(t: integer; c: channel; ch: integer);
procedure poly(t: integer; c: channel);
procedure playsynth(t: integer; view sf: string);

begin ! constructor
end;
```

```
.

begin ! sound
end.
```

A **synth** object can be created as:

```
program p;

joins sound;

var si(synth_out): sound.synth;

begin

   ! output note c in 4th octave at current time, max volume
   si.noteon(1, note_c+octave_4)

end.
```

## O.10 Waveform objects

The functionality of a waveform device is available as a class:

```
module sound;

class wave(int port);

procedure playwave(t: integer; view sf: string);
procedure volwave(t, v: integer);

begin ! constructor
end;

.

begin ! sound
end.
```

A **wave** object can be created as:

```
program p;

joins sound;

var wi(1): sound.wave;

begin
```

```
   wi.playwave("mysound") ! play a waveform file
```

**end**.

## O.11 Exceptions

The following exceptions are generated in **sound**:

| Identifier | Meaning |
|---|---|
| **SequencerNotRunning** | The sequencer is not running |
| **InvalidChannel** | Invalid channel number |
| **InvalidNote** | Invalid note number |
| **InvalidInstrument** | Invalid instrument number |
| **InvalidMonoMode** | Invalid Mono mode number |
| **PlayDefaultOutput** | Must play on default output channel |
| **SynthOutputNotOpen** | Synthesizer output channel not open |

**sound** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **sound** will go back to **sound**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **sound** need only capture the exceptions occurring in the procedure or function that is called.

## O.12 Functions and Procedures in sound

Note that for the method versions of these routines, the synthesizer or wave device port is not specified.

**procedure** starttime;

Start time for sequencer. Starts the sequencer running. If the sequencer is already running, it will be restarted at 0.

Exceptions: None

**procedure** stoptime;

Stop sequencer. Halts the sequencer timer, and releases it.

Exceptions: None

function curtime: integer;

Get current sequencer time. Returns the current sequencer time, in 100 Microsecond counts. The count is guaranteed not to wrap for 24 hours.

Exceptions: **SequencerNotRunning**

function synthout: integer;

Find number of output synthesizers. Returns the total output synthesizers in the system.

Exceptions: None

procedure opensynthout(p: integer);

Open output synthesizer. Opens the output synthesizer by the logical number **p**, where p is 1..**synthout**.

Exceptions: None

procedure closesynthout(p: integer);

Close output synthesizer. Closes the output synthesizer by the logical number **p**.

Exceptions: None

procedure noteon([p: integer;] [t: integer;] c: channel; n: note[; v: integer]);

Start note. Starts a note for synthesizer **p**, in channel **c**, with note n, and 0..**maxint** ratioed volume **v**. If the time **t** is left off, it defaults to 0. If the volume **v** is left off, it defaults to **maxint**. It is not possible to leave the time off and leave the volume present.

Exceptions: **InvalidChannel, InvalidNote, SequencerNotRunning**

procedure noteoff([p: integer;] [t: integer;] c: channel; n: note[; v: integer]);

Stop note. Stops a note for synthesizer **p**, in channel **c**, with note, and 0..**maxint** ratioed volume **v**. **v** is usually ignored on a **noteoff**. If the time **t** is left off, it defaults to 0. If the volume **v** is left off, it defaults to **maxint**. It is not possible to leave the time off and leave the volume present.

Exceptions: **InvalidChannel, InvalidNote, SequencerNotRunning**

procedure instchange([p: integer;] [t: integer;] c: channel; i: instrument);

Change instrument. Changes the instrument assigned to a channel, for output port **p**, at time **t**, for channel **c**, to instrument **i**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, InvalidInstrument, SequencerNotRunning**

procedure attack([p: integer;] [t: integer;] c: channel; at: integer);

Set attack time. Sets the attack time for synthesizer output port p, at time **t**, for channel **c**, to 0..**maxint** ratioed time at. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure release([p: integer;] [t: integer]; c: channel; rt: integer);

Set release time. Sets the release time for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed time at. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure legato([p: integer;] [t: integer]; c: channel; b: boolean);

Set legato. Sets legato mode on or off, for synthesizer output port **p**, at time **t**, for channel **c**, to on/off value **b**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure portamento([p: integer;] [t: integer]; c: channel; b: boolean);

Set portamento. Sets portamento mode on or off, for synthesizer output port **p**, at time **t**, for channel c, to on/off value **b**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure vibrato([p: integer;] [t: integer]; c: channel; v: integer);

Set vibrato. Sets vibrato amount, for synthesizer output port p, at time **t**, for channel **c**, to 0..**maxint** ratioed value **v**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure volsynthchan([p: integer;] [t: integer]; c: channel; v: integer);

Set volume for channel. Sets volume for channel, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **v**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure porttime([p: integer;] [t: integer]; c: channel; v: integer);

Set portamento time. Sets portamento time, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **v**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure balance([p: integer;] [t: integer]; c: channel; b: integer);

Set channel balance. Sets the right left balance for synthesizer output port **p**, at time **t**, for channel **c**, to -**maxint**..**maxint** ratioed value. -**maxint** is full left, **maxint** is full right, and 0 is centered. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure pan([p: integer;] [t: integer]; c: channel; b: integer);

Set channel pan. Sets the right left pan for synthesizer output port **p**, at time **t**, for channel **c**, to -**maxint**..**maxint** ratioed value. -**maxint** is full left, **maxint** is full right, and 0 is centered. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure timbre([p: integer;] [t: integer]; c: channel; tb: integer);

Set timbre. Sets timbre amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **tb**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure brightness([p: integer;] [t: integer]; c: channel; b: integer);

Set brightness. Sets brightness amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **b**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure reverb([p: integer;] [t: integer]; c: channel; r: integer);

Set reverb. Sets reverb amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **r**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure tremulo([p: integer;] [t: integer]; c: channel; tr: integer);

Set tremulo. Sets tremulo amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **tr**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure chorus([p: integer;] [t: integer]; c: channel; cr: integer);

Set chorus. Sets chorus amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **cr**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure celeste([p: integer;] [t: integer]; c: channel; ce: integer);

Set celeste. Sets celeste amount, for synthesizer output port p, at time **t**, for channel **c**, to 0..**maxint** ratioed value **ce**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure phaser([p: integer;] [t: integer]; c: channel; ph: integer);

Set phaser. Sets phaser amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **ph**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure aftertouch([p: integer;] [t: integer]; c: channel; n: note; at: integer);

Set aftertouch. Sets aftertouch amount, for synthesizer output port p, at time **t**, for channel **c**, to 0..**maxint** ratioed value **at**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, InvalidNote, SequencerNotRunning**

procedure pressure([p: integer;] [t: integer]; c: channel; n: note; pr: integer);

Set pressure. Sets pressure amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **pr**. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, InvalidNote, SequencerNotRunning**

procedure pitch([p: integer;] [t: integer]; c: channel; pt: integer);

Set pitch bend. Sets the pitch "bend", or change amount, for synthesizer output port **p**, at time **t**, for channel **c**, to -**maxint**..**maxint** ratioed value **pt**. **pt** value is -**maxint** for full down range, **maxint** for full up range, and 0 for neutral (on note) pitch. The amount of pitch range is set by the ptichrange procedure, and defaults to one note down and one note up. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure pitchrange([p: integer;] [t: integer]; c: channel; v: integer);

Set pitch bend range. Sets the total amount of pitch change that can be reached by the pitch command, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**maxint** ratioed value **v**. 0 disables the pitch command, and **maxint** allows it to reach all 128 notes of MIDI. Note that increasing the range of the pitch command decreases its resolution. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure mono([p: integer;] [t: integer]; c: channel; ch: integer);

Set mono mode. Sets mono mode for synthesizer output port **p**, at time **t**, for channel **c**, for the number of channels **ch**. See MIDI specification for details. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning, InvalidMonoMode**

procedure poly([p: integer;] [t: integer]; c: channel);

Set polyphonic mode. Sets polyphonic mode for synthesizer output port **p**, at time **t**, for channel **c**. Reverses the effect of a mono operation. If the time **t** is left off, it defaults to 0.

Exceptions: **InvalidChannel, SequencerNotRunning**

procedure playsynth([p: integer;] [t: integer]; sf: string);

Play MIDI synthesizer file. Plays the MIDI instructions from the file by the name in **sf**, for output synthesizer **p**, at time **t**. If the time **t** is left off, it defaults to 0.

Exceptions: **SequencerNotRunning, PlayDefaultOutput, SynthOutputNotOpen**

function waveout: integer;

Find number of waveform output files. Returns the total number of waveform files in the system.

Exceptions: None

procedure openwaveout(p: integer);

Open waveform device. Opens the logical waveform device **p**, where **p** is 1..**waveout**.

Exceptions: None

procedure closewaveout(p: integer);

Close waveform device. Closes the logical waveform device **p**.

Exceptions: None

procedure playwave([p: integer;] [t: integer]; sf: string);

Play waveform file. Plays the waveform file by the name **sf**, for output waveform device **p**, at time **t**. If the time **t** is left off, it defaults to 0.

Exceptions: **SequencerNotRunning**

procedure volwave(p, t, v: integer);

Set waveform volume. Sets the output waveform device volume for logical device **p**, at time **t**, to 0..**maxint** ratioed value **v**. If the time **t** is left off, it defaults to 0.

Exceptions: None

# P Annex: Networking Library

**network** gives Pascaline the ability to transfer data over a network such as the internet. It does this by connecting ISO 7185 Pascal files to network resources. Because of the use of standard file mechanisms, few added calls are needed.

To open a new network connection, **opennet** is used. To close network connections, the standard Pascaline **close** is used. **opennet** uses an address/port pair to indicate the network address of the server, and the port within the server. The address of a server, as determined from its name in characters, is found with **addrnet**.

When a remote network port is opened, **network** treats the connection as a pair of communications channels, one going to, and one coming from, the remote resource. This makes it easier to use the standard idea in Pascal of a file having a read or write mode.

## P.1 Exceptions

The following exceptions are generated in **network**:

| Identifier | Meaning |
|---|---|
| **Cannot initialize** | Cannot initialize network access |
| **InvalidFile** | Invalid file handle |
| **CannotResetOrRewriteNetwork** | Cannot apply reset or rewrite to network file |
| **CannotPositionNetwork** | Cannot apply position to network file |
| **CannotFindLocationNetwork** | Cannot apply location to network file |
| **CannotFindLengthNetwork** | Cannot apply length to network file |
| **EndEncountered** | End of network file encountered |
| **FileInUse** | File for network access is already in use |
| **CannotWriteToInput** | Cannot write to input side of network pair |

**network** establishes a series of exception handlers for each of the above exceptions during startup. Exceptions not handled by a client program of **network** will go back to **network**, then print a message specific to the error, then the general exception will be thrown.

Not all procedures and functions throw all exceptions. See each procedure or function description for a list of exceptions thrown. A client of **network** need only capture the exceptions occurring in the procedure or function that is called.

Note that **network** defines several new exceptions for standard file operations such as **read**/**readln**, **write**/**writeln**, close and others.

## P.2 Functions and Procedures in network

procedure opennet(var infile, outfile: file; addr: lcardinal; port: lcardinal);

The server is indicated by a logical address number **addr**, whose exact meaning and format is dictated by the network itself. A logical port number **port** selects which resource within the server is being accessed. For the internet, this is a fixed constant that gives the exact service being asked of the far server.

When a network link is opened, the input coming from the far server is connected to the **infile** of the **opennet** call, and the output to be sent to the far side is connected to the **outfile**. These files establish two way communication with the far server.

Exceptions: **FileInUse**

procedure addrnet(name: string; var addr: lcardinal); forward;

**addrnet** takes the logical name of a server in string **name** and finds the address number **addr** for it. Such names are formatted according to the needs of the network. In the internet, such names consist of the characters:

Network connections are ended by the end of the program, or by using the standard file **close** procedure on either of the in or out files of the connection.

Exceptions: None