HSPascal

2.0

HSPascal
Christen Fihl
http://HSPascal.Fihl.net

Indhold

1. HSPascal **1**
1.1 Structure of this paper 1

2. Language Elements **2**
2.1 Basic Symbols 2
2.2 Separators 2
2.3 Comments 2
2.4 Reserved Words 3
2.5 Identifiers 3
2.6 Numbers 3
2.7 Strings 4
2.8 Constant Declarations 4

3. Types **6**
3.1 Type declaration part 6
3.2 Type declaration 6
3.3 Simple Types/Ordinal Types 6
3.3.1. Integer 7
3.3.2. Boolean 7
3.3.3. Char 7
3.3.4. Enumerated 8
3.3.5. Subrange 9
3.3.6. Real (Single) 9
3.4 Structured Types 9
3.4.1. Array 10
3.4.2. Record 10
3.4.3. Set 12
3.5 String 12
3.6 Pointer 13

4. Variables **14**
4.1 Variable declaration part 14
4.2 Variable declaration 14
4.3 Variables in different places 14
4.4 Use of Variables 14
4.5 Qualifiers 15
4.5.1. Arrays and String indexes 15
4.5.2. Records 15
4.5.3. Pointers 16
4.6 Variable Typecast 16

5. Expressions **17**
5.1.1. Operator precedence 17
5.2 Arithmetic Operators 18
5.2.1. Unary arithmetic operations 18
5.2.2. Binary arithmetic operations 18
5.3 Boolean Operators 19
5.3.1. Boolean operations 19
5.3.2. Boolean evaluation 19
5.4 Logical Operators 19
5.4.1. Logical operations 19
5.5 Set Operators 20
5.6 Relational Operators 20

5.6.1. Comparing Ordinals 20
5.6.2. Comparing Reals 20
5.6.3. Comparing Strings 20
5.6.4. Comparing Pointers 21
5.6.5. Comparing Sets 21
5.6.6. Testing Set Membership 21
5.7 The @ Operator 21
5.8 Function Calls 21
5.9 Set Constructors 22
5.10 Value Typecast 22

6. Statements 23
6.1 Simple Statements 23
6.2 Structured Statements 23
6.3 Assignment 23
6.4 Procedure Statements 24
6.5 Goto Statements 24
6.6 Compound Statements 24
6.7 BreakStatement 25
6.8 ContinueStatement 25
6.9 If Statements 25
6.10 Case Statements 25
6.11 For Statements 26
6.12 Repeat Statements 27
6.13 While Statements 27
6.14 With Statements 27
6.15 Assembler Statement 28

7. Procedures and Functions 29
7.1 Forward 29
7.2 Parameters 30
7.2.1. Value Parameters 31
7.2.2. Variable Parameters 31
7.2.3. Const Parameters 31
7.2.4. Untyped Variable Parameters 31
7.3 Procedure 32
7.4 Function 32

8. Programs, Units and Procedures 33
8.1 Blocks 33
8.2 Program Syntax 33
8.3 Unit Syntax 34

9. Scope and Activations 36
9.1 Scope for Units 36
9.2 Scope for Standard Identifiers 36
9.3 Activations 36

10. Resources 37
10.1 Plain resource 37
10.2 Menu ressource 38
10.3 Form ressource 38

11. Inline Assembler 40
11.1 Labels 41
11.2 ASSEMBLER procedures and functions 41
11.4 Expressions 42

11.5 Registers 43
11.6 Addressing modes 43
11.7 Operators 43
11.8 Limitations 44
11.9 PalmOS, System calls 44
11.9.1. Functions calls: 44
11.9.2. Callback (EventHandler, 1 parameter) 44
11.9.3. Callback (more parameters) 45

12. Inside HSPascal 47
12.1 Memory Layout 47
12.3 Data formats 47
12.3.1. Integer Types 47
12.3.2. Char Types 47
12.3.3. Enumerated Types 47
12.3.4. Boolean Types 47
12.3.5. String Types 47
12.3.6. Real Types 48
12.3.7. Pointer Types 48
12.3.8. Set Types 48
12.3.9. Array Types 48
12.3.10. Record Types 48
12.4 Calling Conventions 48
12.4.1. Value Parameters 50
12.4.2. Variable Parameters 50
12.4.3. Const Parameters 50
12.4.4. Function Results 50

13. Compiler Directives 51
13.1 Switch directives 51
13.1.1. Debug Information 51
13.1.2. Line information, for debugger singlestepping 51
13.1.3. Symbol table output 51
13.2 Source code control 51
13.2.1. $Define, $Undef 51
13.2.2. $SetDefault 52

14. Map file layout 53
14.1.1. Test program 53
14.1.2. Assembler code 53
14.1.3. Map file generated 53

15. Command Line Compiler 55
15.1 $ Switch Option 55
15.2 Configuration File 55

16. Compiler and Runtime Errors 56
16.1 Compiler errors 56

17. HSPascal versus other Pascal Compilers 60
17.1 HS Pascal versus Turbo Pascal for the PC 60
17.2 HSPascal versus ANS Pascal 60
17.3 Extensions to ANS Pascal 61

18. Pascal for C programmers 62
18.1 Constants 62
18.2 Types 62
18.2.1. Enumerated values 62

18.2.2. Arrays ... 63
18.2.3. Pointers ... 63
18.2.4. Structures / Records ... 63
18.3 Procedures / Functions ... 63

19. FAQ ... 64
19.1 How to make a new program icon from a Windows bitmap ... 64
19.2 How to include a Windows bitmap into a program ... 64
19.3 How to load a Bitmap (Tbmp #2000) ... 64
19.4 Draw a string value ('Hello World') into a screenfield (id2000) ... 64
19.5 Draw an integer value (123) into a screenfield (id2000) ... 65
19.6 How to select items from a StringList ... 65

20. Booklist, literature ... 66

1. HSPascal

HSPascal, aka. High Speed Pascal, is designed, programmed and sold by Christen Fihl, Denmark.

1.1 Structure of this paper

The more formal BNF notations of the Pascal language are spread around this manual. And it is show bottom up, as the smallest items are shown first, starting with letters and digits, ending up in a full program.

2. Language Elements

2.1 Basic Symbols

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

These are the basic building blocks:

```
Letter =
  A to Z, a to z and underscore '_'.
Digit =
  The ten digits from 0 to 9.
HexDigit =
  The normal digits, A to F and a to f.
Blank =
  All control characters and the space characters.
Special =
  + - * / = < > ( ) [ ] { } , . ; : ' ^ @ $ #
  <>   <=   >=   :=   ..   (*   *)   (.   .).
```

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

These symbols have the same meaning:

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

2.2 Separators

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

2.3 Comments

Comments can be inserted anywhere where you are allowed to insert a blank or newline. A comment is bounded by { and } or by (* and *). A comment can span over several lines.

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

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

2.4 Reserved Words

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

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

2.5 Identifiers

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

```
Identifier =
  Letter { Letter | Digit }
```

2.6 Numbers

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

Some examples:

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

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

Using the normal engineering notation format makes Numbers for reals.
1.23e4 is the same as 12300.00 or 12300.

```
UnsignedNumber =
  UnsignedInteger | UnsignedReal.

UnsignedInteger =
  DigitSequence | "#" HexDigitSequence.

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

ScaleFactor =
  [ Sign ] UnsignedInteger.

Sign =
  "+" | "-"

DigitSequence =
   Digit { Digit }.
```

```
HexDigitSequence=
  HexDigit { HexDigit }.

SignedNumber =
  SignedInteger | SignedReal.

SignedInteger =
  [ Sign ] UnsignedInteger.

SignedReal =
  [ Sign ] UnsignedReal.
```

2.7 Strings

A character string is a sequence of characters enclosed in single quotation marks.
A quotation mark can itself be included by entering it as twice.

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

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

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

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

Some examples:

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

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

2.8 Constant Declarations

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

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

```
ConstantDeclaration =
  Identifier "=" Constant ";".

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

ConstantIdentifier =
  Identifier.
```

These constants are predefined in HSPascal:

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

3. Types

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

3.1 Type declaration part

Type declarations typically appear in procedures and programs. The type declaration follows right after the "type" keyword. In HSPascal there can be more than one "type" declaration block, and it can be mixed with constant, label, procedure, function and variable declarations.

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

Example:

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

3.2 Type declaration

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

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

```
Type
  TMyStr: String;
  TMoney: Real;
```

3.3 Simple Types/Ordinal Types

```
SimpleType =
      OrdinalType | RealType
```

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

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

Ordinal types can be worked on using these five routines:

- Ord	returns the ordinality of a value
- Succ	returns the value right after the one given
- Pred	returns the value just before the one given
- Inc	increments the value (default to next value)
- Dec	decrements the value

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

Using typecasting can make the opposite function of Ord.
The boolean value True=Boolean(1).

Do not make constructs like following:

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

Integer, Boolean, Char, Enumerated and subranges are all of ordinal type.

3.3.1. Integer

HSPascal gives you four predefined integer types: ShortInt, Byte, Integer and LongInt. They are defined as:

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

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

3.3.2. Boolean

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

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

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

3.3.3. Char

The Char type gives 256 different characters values. The character set used is the one used on the Palm. The biggest problem you will get with most character-sets is the way they sort national characters. You cannot easily sort characters if they use 8-bit characters as all national letters do.

This always holds on the Palm:

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

3.3.4. Enumerated

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

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

Examples of enumerated types:

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

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration.

```
 Ord(Club)    =0
 Ord(Diamond) =1
```

Enumerated types with explicitly assigned ordinality (as in C#).

You can override this by explicitly assigning ordinalities to some or all of the values in the enumeration declaration. Do this by follow its identifier with = integervalue. For example (from HSSys2.pas),

```
 Type
   SysTrapNumber=(
     sysTrapMemInit =$A000,
     sysTrapMemInitHeapTable,
```

defines a SysTrap (Palm function) sysTrapMemInit with value $A000, sysTrapMemInitHeapTable = $A001, and so on.

It is possible to give same value to more than one element. The declaration

```
 type Enum = (e1, e2, e3 = 1);
```

is the same as

```
 type Enum = (e1=0, e2=1, e3 = 1);
```

and then e1 and e3 has the same value!

3.3.5. Subrange

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

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

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

That’s it for ordinal types. The last type in SimpleType is RealType.

3.3.6. Real (Single)

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

All real types are predefined and cannot be extended by new user types.
One kind of reals is implemented: Single. Doubles can be declared, but cannot be used in calculations without using MathLib manually.

Note: The type Real is the same as Single.

Single (and doubles) are IEEE compatible.
Singles are very fast to move around in memory because they only take four bytes which can be done in one assembler line.

This is what you get from the reals:

Type	Range	Usefull digits	
Single = Real	3.4e-38..3.4e38	7-8	Can be used right away in HSPascal
Double	1.7e-308..1.7e308	15-16	Needs special coding using MathLib

3.4 Structured Types

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

The components of a structure may itself be structured.

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

UnpackedStructuredType =
  ArrayType |
  RecordType |
  SetType

StructuredTypeIdentifier =
  TypeIdentifier.
```

3.4.1. Array

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

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

  IndexType =
    OrdinalType.

  ComponentType =
    Type.
```

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

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

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

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

Packed array[1..N] of char has special properties not given to other array types: It can be assigned to a string.

Note: When you write array[7..9] of integer; you actually create a new subrange type NoName=7..9;, it just does not have a name.

3.4.2. Record

Array type had one element type but could contain a fixed number of them.
Record types can contain a fixed number of components with different types.

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

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

  FixedPart =
    RecordSection { ";" RecordSection }.

  RecordSection  =
    IdentifierList ":" Type.

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

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

   VariantSelector =
     [ TagField ":" ] TagType.

   TagType =
     OrdinalTypeIdentifier.

   TagField =
     Identifier.
```

The IdentifierList names each element, as in a variable declaration.

A normal record without variant fields looks like this:

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

With a variant field it looks like this:

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

An example of its use:

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

One use of variant parts is for data conversion:

```
 Program VariantTest;
 Uses Crt;
 var
   CV: record
         case integer of    {Does not require any space}
         4: (L: LongInt);
         1: (B1,B2,B3,B4: Byte);
       end;
 begin
   CV.L:=$01020304;
   writeln(CV.B1);      //=$01
   writeln(CV.B2);      //=$02
   writeln(CV.B3);      //=$03
   writeln(CV.B4);      //=$04
 end.
```

3.4.3. Set

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

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

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

Example:

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

3.5 String

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

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

   SizeAttribute =
     UnsignedInteger.
```

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

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

Examples of comparison:

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

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

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

3.6 Pointer

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

```
PointerType =
  "^" BaseType.

BaseType =
  OrdinalType.
```

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

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

4. Variables

4.1 Variable declaration part

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

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

Example:

```
var
  Counter,
  M,N,O   : Integer;
  C1, C2  : Color;
  BigSet  : Set of Byte;
```

4.2 Variable declaration

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

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

4.3 Variables in different places

Global variables are those in programs and units, but outside of procedure and functions. The total size of globals variables are limited to 32KByte.

No single variable can be larger than 32KByte.

Variables in procedure and functions are created when the procedure is activated and removed right after the activation is terminated. Each procedure can only have 32 KBytes of local variables, but the total size of the stack can be as large as memory allows. Using too much stack space might stop the program with a runtime error if required. On the Palm, the stack space might be very limited, like 5 KByte in total.

Dynamic variables are created with New and removed again with Dispose.

4.4 Use of Variables

Using a variable is called referencing a variable.

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

4.5 Qualifiers

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

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

```
   Rect
   Person
```

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

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

4.5.1. Arrays and String indexes

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

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

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

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

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

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

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

4.5.2. Records

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

```
   FieldDesignator =
     "." FieldIdentifier.
```

Example:

```
   Person.Name
   Person.Name.First
```

4.5.3. Pointers

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

Example:

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

4.6 Variable Typecast

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

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

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

5. Expressions

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

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

5.1.1. Operator precedence

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

Expression syntax:

```
Expression =
  SimpleExpression [ RelationalOperator SimpleExpression ].

SimpleExpression =
  [ Sign ] Term { AddingOperator Term}

Term =
  Factor { MultiplyingOperator Factor }

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

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

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

ElementDescription =
  Expression [ ".." Expression ].
```

5.2 Arithmetic Operators

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

5.2.1. Unary arithmetic operations

Operator	Operation	Type of Operand	Type of Result
+	none	integer/real	integer/real
-	none	integer/real	integer/real

5.2.2. Binary arithmetic operations

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

If one of the operands are of real type then the type of the result is always real; otherwise if one of the operands is of type longint then the result is also of longint type else the result is of integer type.

The rules used by div and mod are:

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

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

5.3 Boolean Operators

5.3.1. Boolean operations

Operator	Operation
not	negates the boolean value (monadic)
and	takes the logical and
or	takes the logical or
xor	logical xor

The not operator negates
The and operator takes the logical and.
The or operator
The xor operator takes the logical xor.

5.3.2. Boolean evaluation

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

5.4 Logical Operators

5.4.1. Logical operations

The operators not, and, or and xor can also be used as bitwise operators. Bitwise operators work on all bits in the operand(s). Besides the operators from boolean operators there are also the two new operators shl and shr.

Operator	Operation
not	makes bitwise negation (monadic)
and	makes bitwise and
or	makes bitwise or
xor	makes bitwise xor
shl	shifts bitwise to the left
shr	shifts bitwise to the right

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

5.5 Set Operators

Three set operands exists:
+ union
- difference
* intersection

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

Set union (a+b) returns all members that are **either** in a **or** in b **or in both**.
Set difference (a-b) returns all members **in** a but **not in** b.
Set intersection (a*b) returns all members that are in **both** a **and** b.

5.6 Relational Operators

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

5.6.1. Comparing Ordinals

Operands applicable: =, <>, <, <=, >, >=.
Each ordinal operand must be of compatible type.
The result is the mathematical relation of their ordinalities.

Example:
```
   Red < Green
```

5.6.2. Comparing Reals

Operands applicable: =, <>, <, <=, >, >=.
Take care when comparing reals to be equal and not-equal.

5.6.3. Comparing Strings

Operands applicable: =, <>, <, <=, >, >=.
All strings are compatible so all strings can be compared.
A char variable can be treated as a string with actual length 1.

Examples of comparison:
```
   'Hello' = 'Hello'
   'Hello' < 'Hello World'
   'Hello' > ''
   'Hello' < 'hello'
   'Hello' <= 'hello'
   'Hello' <> 'hello'
```

Strings are compared pure binary. This can be a problem when sorting strings which include national characters.

5.6.4. Comparing Pointers

Operands applicable: =, <>.
Each pointer compared must be of compatible types. Two pointers are only equal if they point to the same element.

5.6.5. Comparing Sets

Operands applicable: =, <>, <=, >=.
Both operands must be of compatible types. If a and b are sets, then:

- a=b	is true if every member in a is a member of b and visa versa.
- a<>b	is true if no member in a is a member of b and visa versa.
- a<=b	is true if every member in a is also a member of b.
- a>=b	is true if every member in b is also a member of a.

5.6.6. Testing Set Membership

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

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

Example:
```
   B:= 2 in [1..4];   //True
   B:= 2 in [7,9,13]; //False
```

5.7 The @ Operator

The @ operator creates a pointer with a type like that of nil. The pointer can be assigned to any pointer variable. The operator works on variables, procedure and functions.

Because the type returned is compatible with all pointers, you do not get any type checking from the compiler. You can easily (by accident) assign the address of a char to an integer pointer.
Only use @ when you absolutely have to, and take care when using it, or start using C.

5.8 Function Calls

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

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

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

Examples
```
   N:=Max(i,j);
   R:=Random;
```

5.9 Set Constructors

Set values are created by writing expressions within brackets.

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

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

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

Examples:
```
   Set1:= [];
   Set1:= [May,June];
   Set2:= ['0'..'9','a'..'z'];
```

5.10 Value Typecast

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

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

Examples:
```
   Ch:=Char(65);         {The same as chr(65)}
   B:=Boolean(1);        {Same as True}
   L:=Longint(@Buffer);
```

6. Statements

Statements are said to be executable.

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

6.1 Simple Statements

The simple statement is a self contained statement.

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

  EmptyStatement = . (Nothing)
```

6.2 Structured Statements

Structured statements contains other statements which have to be executed in:

• sequence	compound statements,
• conditionally	if and case statements,
• repeatedly	repeat, while and for statements or
• with statements	within an expanded scope.

```
StructuredStatement =
  CompoundStatement |
  IfStatement |
  CaseStatement |
  ForStatement |
  RepeatStatement |
  WhileStatement |
  WithStatement |
  GotoStatement |
  BreakStatement |
  ContinueStatement |
  AsmStatement.
```

6.3 Assignment

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

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

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

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

6.4 Procedure Statements

```
ProcedureStatement =
  procedureIdentifier [ ActualParameterList ]
```

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

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

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

6.5 Goto Statements

Statements can be prefixed with Labels which enable you to jump around in the program with goto's. Try to avoid goto's and use the repeat, while and the other high-level statements instead.

```
GotoStatement =
  "goto" Label.
```

Two rules must be remembered:
- The goto statement and the corresponding label must be in the same block.
- Do not jump into a structured statement. **for** loops does some initial calculations, that cannot be skipped.

6.6 Compound Statements

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

```
CompoundStatement =
  "begin" StatementList "end".

StatementList =
  Statement { ";" Statement }.
```

Note that it is legal to write as many semicolons as you like.
The last statement "executed" could be empty, leaving only the ";" back on the text line.

6.7 BreakStatement

```
BreakStatement =
  "break"
```

Used inside repeating loops ("for", "while", "repeat") and will force the loop to stop its execution.

6.8 ContinueStatement

```
ContinueStatement =
  "continue"
```

Used inside repeating loops ("for", "while", "repeat") and will force the loop execution to jump to the last line in the (compound) statement.

Example (very special):

```
repeat
  continue
until true
```

This loop will **not** continue forever. One could think the statement `continue` would make the execution jump to the first line (`repeat`), but it is the last (`until true`) that is the next to execute.

6.9 If Statements

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

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

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

Examples:

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

6.10 Case Statements

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

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

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

```
      { ";" }
      "end".

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

    ConstantElement =
      Constant [ ".." Constant ].

    ElseClause =
      "else" StatementList.
```

Example:

```
    Case NextSymbol of
    '+': Expression:=n+Factor;
    '-': Expression:=n-Factor;
    Else
      writeln('Parsing error');
      exit
    end;
    case Month of
    1,3,5,7,8,10,12: Days:=31;
    2: if Leap(Year) then Days:=29 else Days:=28;
    else Days:=30;
    end;
```

6.11 For Statements

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

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

    ControlVariable =
      VariableIdentifier.

    InitialValue =
      Expression.

    FinalValue =
      Expression.
```

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

The control variable takes all values from initial value to final value including both. When using the "to" format the control variable increments by one for each time the loop executes the statement. When using "downto", the control variable gets decremented. The statement does not execute at all if the initial variable is greater then the final value when using the "to" format, or the opposite when using the "downto" format.

It is not legal to change the control variable in the loop. This is not checked in HSPascal. When the last loop has finished the control variable becomes undefined. If you jump out of the loop using a goto or break statement, the control variable is still valid (containing the last value used).

Examples:
```
For n:=1 to 30000 do ;  {Nothing}
For n:=1 to 9 do
  if Odd(n) then writeln(i2s(n)+' is an odd number')
  else writeln(i2s(n)+' is an even number');
```

6.12 Repeat Statements

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

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

Examples:
```
repeat write('*') until Terminated;
```

6.13 While Statements

The while loop works almost as the repeat loop except that the expression is evaluated first.

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

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

6.14 With Statements

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

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

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

Examples:

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

This is equivalent to

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

This is equivalent to

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

Another example:

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

Note that this is textually equal to writing

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

but the program will behave in quite another way.

6.15 Assembler Statement

```
 AsmStatement=
   ASM [Label:] AsmStmt {Separator AsmStmt} end
```

Example:

```
Asm
  ADDQ.W   #1,N
End;
```

See more in Chapter 11, Inline Assembler

7. Procedures and Functions

This section describes how to declare procedures and functions.

A procedure statement activates procedures while a function is activated when used in an expression that contains its function name.

Procedures and functions are defined as:

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

FunctionDeclaration =
  FunctionHeading ";" FuncProcBody ";".

FuncProcBody =
  Block |
  Forward |
  Assembler .

ProcedureHeading =
  "procedure" Identifier [ FormalParameterList ].

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

ResultType =
  TypeIdentifier | "string".
```

If the procedure body is declared as a block, all the declaration is done together. Alternatively, it may be declared as a forward declaration, either using the forward declaration or as a header definition in the interface part of a unit.

Examples:

```
procedure HexDump(Val,Format: Integer); forward;
Function Power10(N: Integer): Real;
procedure HexDump;
```

Procedure and functions can also be made using assembler. See how in chapter: 11.2, ASSEMBLER procedures and functions.

7.1 Forward

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

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

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

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

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

Another example, a full unit source:
```
Unit SmallTest;

Interface

Uses Crt, HSUtils;

procedure MyTest(A: Integer);   //forward without the word "forward"

Implementation

procedure MyTest(A: Integer);
begin
  Writeln(i2s(A))
end;

end.
```

7.2 Parameters

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

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

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

  ParameterType =
    TypeIdentifier | "string".
```

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

7.2.1. Value Parameters

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

The actual parameter must be an expression that does not contain any file type elements. If the parameter type is string, the formal parameter has a type of String[64]. (System dependent)

The actual parameter must be assignment compatible with the formal parameter.

7.2.2. Variable Parameters

Variable parameters are a parameter declaration preceded by **var** and followed by a type. A formal variable parameter is located right on top of its actual parameter, so each time the formal parameter gets changed, the actual parameter is also changed.

If the parameter type is string, the formal parameter has a type of String[64] (System dependent), and the actual parameter must also have a length of 64.

The type of the actual parameter and the type of the formal parameter must be identical.

7.2.3. Const Parameters

Const parameters are a parameter declaration preceded by **const** and followed by a type. Works like **var**, but some restrictions does apply like you are not allowed to change the variable.

7.2.4. Untyped Variable Parameters

Untyped variable parameters are a parameter declaration preceded by **var** but without any type following it.

The actual parameters passed can be of any type.
The formal parameters are typeless. They can only be used as typeless pointers or by using variable typecasting as in this example:

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

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

```
   B:=Equal(TestVar[10],TestVar2[20],SizeOf(LongInt));
   B:=Equal(TestVar[10],TestVar2[20],SizeOf(LongInt)*10);
```

Be careful when using untyped variables. It is similar to calling a procedure in the C language. You do not get much checking.

7.3 Procedure

A procedure is activated in a procedure statement like: WriteInt (123).

Here is what a normal procedure might look like:

```
   procedure WriteInt(N: Integer);
   var
     S: String[6];
   begin
     Str(N,S);
     writeln(S)
   end.
```

A procedure can modify global variables, but it is discouraged. This is called "side effect".

7.4 Function

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

A normal function could look like this:

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

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

8. Programs, Units and Procedures

8.1 Blocks

All code pieces used to form a Pascal program have a name. Every one of these pieces are collected into a block. This can be a procedure, a function, a program or a unit. In each block, new identifiers can be declared, which are local to the block. The only exception is a unit, declaring identifiers for use by others.

```
Block =
  DeclarationPart StatementPart.

DeclarationPart =
  { LabelDeclarationPart |
  ConstantDeclarationPart |
  TypeDeclarationPart |
  VariableDeclarationPart |
  ProcedureDeclarationPart |
  FunctionDeclarationPart |
  ResourceDeclarationPart }.

LabelDeclarationPart =
  "label" Label [ "," Label ] ";"

Label =
  Identifier | DigitSequence

ConstantDeclarationPart =
  "const" ConstantDeclaration
  { ConstantDeclaration }.

TypeDeclarationPart =
  "type" TypeDeclaration
  { TypeDeclaration }.

VariableDeclarationPart =
  "var" VariableDeclaration
  { VariabelDeclaration }.

ProcedureDeclarationPart =
  ProcedureDeclaration.

FunctionDeclarationPart =
  FunctionDeclaration.

ResourceDeclarationPart =
  ResourceDeclaration
```

See Chapter 10, Resources for ResourceDeclaration.

8.2 Program Syntax

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

```
Program =
  [ ProgramHeading ] [ UsesClause ] Block ".".

ProgramHeading =
  "program" Identifier [ ProgramParameterList ].

ProgramParameterList =
  "(" IdentifierList ")".

UsesClause =
  [ "uses" IdentifierList ].
```

The identifier after "program" does nothing. The final file for the program is named as the source file itself.

8.3 Unit Syntax

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

```
Unit =
  "unit" Identifier ";"
  InterfacePart
  ImplementationPart
  InitializationPart "."

InterfacePart =
  "interface" UsesClause
  { ConstantDeclarationPart |
  TypeDeclarationPart |
  VariableDeclarationPart |
  ProcedureHeadingPart |
  FunctionHeadingPart }.

ProcedureHeadingPart =
  ProcedureHeading ";" [ InlineDirective ";" ].

FunctionHeadingPart =
  FunctionHeading ";" [ InlineDirective ";" ].

ImplementationPart =
  "implementation"
  { LabelDeclarationPart |
  ConstantDeclarationPart |
  TypeDeclarationPart |
  VariableDeclarationPart |
  ProcedureDeclarationPart |
  FunctionDeclarationPart }.

InitializationPart =
  "end" | StatementPart.
```

There is no label declaration in the interface part.
The implementation part must implement all the procedures and functions defined in the

interface part.
Everything not defined in the interface part is private to the unit.

The code entered in the initialization part is executed right before the first begin in the program module. They are executed in the same order as used.

A full unit source as example:
```
Unit SmallTest;

Interface

Uses Crt, HSUtils;

procedure MyTest(A: Integer);   //forward without the word "forward"

Implementation

procedure MyTest(A: Integer);
begin
  Writeln(i2s(A))
end;

end.
```

And a full program using it:
```
Program DoSmallTest;

Uses SmallTest;

begin
  MyTest(123)
end.
```

9. Scope and Activations

The scope rules determine where an identifier can be used.

- An identifier cannot be used, before it has been defined. "Before" means reading the source text from top to bottom.
- An identifier cannot be used after the block in which it is defined ends.
- An identifier cannot be redefined in the same block.
- An identifier can be redefined in an enclosed block.
- An identifier can be reused in a record structure. This does not hide the first identifier from being used.

9.1 Scope for Units

Each unit used in a program or unit opens up a new scope. An identifier can be redefined in each of these units, and only the last seen will be used.

9.2 Scope for Standard Identifiers

All identifiers defined by HSPascal are defined in the System units, named HSSys?.pas. These units are always the first one used. As the normal scope rules for units applies, all identifiers can therefore be redefined.

9.3 Activations

When a procedure is called, it is activated, and it gets an activation. A lot can be said about what an activation contains, but the most interesting is where the local variables are.

Each local variable get new space each time a procedure is called.

Each time a procedure calls itself (recursively or mutually recursively) it gets new space for the locals. Each of the activations, of course, knows where each of its variables for its own activation is.

10. Resources

Palm specific resources can be included by compiler directives like {$R *.bin}.
Another way is to specific them inline in the source. An example (from the Cells.pas example) making a form, a menu with 2 dropdown menus and a dialog follows and finally a version information:

```
 Resource
   MainForm=(ResTFRM,,1000,(0,0,160,160),0,0,MenuRes,
             (ResFTitle,,'Cells')  );
   MenuRes=(ResMBAR,,,
     (ResMPUL,,(6,14,85,88),(4,0,35,12),'Game',
       MenuNew   =(,'G','New Game'),
       MenuLoad  =(,'1','Load #1'),
       MenuSave  =(,'a','Save #1'),
       MenuQuit  =(,'X','Exit')),
     (ResMPUL,,(90,14,60,33),(88,0,30,12),'Help',
       MenuHelp  =(,'H','Help'),
       MenuRules =(,'R','Rules'),
       MenuAbout =(,'A','About')));
   Help1=(ResTalt,,,0,0,0,'Cells - Help',
     'Cells is a port of the popular "Game of '+
     'Life" invented in 1970 by John Conway.'#10#10+
     'In an automata universe, cells follow very basic '+
     'rules to survive, be born '+
     'or die in a turn based simulation.', 'Ok');
   (ResTVER,,1000,'1.2');
```

Resource gives you both the definition, and a constant number to use later on when requesting the resource. "MenuRes" above will give the next unused resource id. But no matter what it happens to be, MenuRes is the value used.
See Appendix "Resources" for exact resource structures.

The general format is:
"(" ResourceType , ResourceID#, params ")"
All ressource types are named as ResXXXX, where XXXX is the four-letter ressource name or FormXXXX for form resources.

10.1 Plain resource

Type	Fields	
RestAIN	, id, ’string’	Application Icon Name (in the string)
RestSTR	, id, ’string’	A single string (in the string)
RestSTL	, id, ’aString’, (’str2’,…’strN’)	StringList
Restver	, id, ’string’	Version string. Id defaults to 1000
Respref	, id, stacksize	Specify wanted stacksize. Id defaults to 1
ResTRAP	, id, number	Trap number ressource
ResTalt	, id, AlertType, HelpID, Default#, ’Title’, ’Message’, ’Button1’, ’’Button2’, ’Button3’,…	Alert dialogs. At least 3 strings must be given.

ResRaw	TYPE, id, string	Include literal data from a string (string) with any type (TYPE) and any id (id)
ResFile	TYPE, id, filename	Include literal data from a file (filename) with any type (TYPE) and any id (id)

10.2 Menu ressource

A menu consists of a ResMBAR with one or more pull down menus (ResMPUL), like "File", "Edit" etc. Each ResMPUL has some items, ResMItem.
The whole structure for a single menu is defined recursive in one big block. But still easy to read.

Type	**Fields**	
ResMBAR	, id, ResMPUL	MPUL is a comma delimited sequence of following
ResMPUL	(LTWH), (LTWH), ’string’, *Item’s*	LTWH1 is position of topline text. LTWH2 is position of items as listed in Mitem’s.
Item’s	id, ’K’, ’string’	’K’ is shortcut key. string is the label

LTWH are the 4 numbers (inside parenthesis) for the Left, Top, Width and Height.
Example:

```
MenuRes=(ResMBAR,,
  (ResMPUL,(6,14,85,88),(4,0,35,12),'Game',
    MenuNew   =(,'G','New Game'),
    MenuLoad  =(,'1','Load #1'),
    MenuSave  =(,'a','Save #1'),
    MenuQuit  =(,'X','Exit')),
  (ResMPUL,(90,14,60,33),(88,0,30,12),'Help',
    MenuHelp  =(,'H','Help'),
    MenuRules =(,'R','Rules'),
    MenuAbout =(,'A','About')));
```

10.3 Form ressource

A form consists of a ResTFRM with some (visual) items placed inside it like Buttons, edit fields, picklists, etc. The whole structure is defined recursive in one big block.

Type	**Fields**	
ResTFRM	, id, (LTWH), DefaultButton#, Help#, Menu#, *items*	LWTH is the screen position of the form. DefaultButton=0 if none, Help#=0 if none, Menu#=0 if none. Items are one or more of the following
Items…		
FormTitle	’string’ Note: There is no id !	Title for the form
FormButton	, id, (LTWH), Attr, Font, Grp, ’ButtonText’	Font=0 or one from palmFont(=stdFont..ledFont,…) Group=0,..
FormLabel	, id, (LT), Attr, Font, ‘label’	Label
FormField	, id, (LTWH), Attr,	Field

	MaxLen, Font	
FormPushButton	, id, (LTWH) , Attr, Font, Grp, ‘Text’	Push Button
FormCheckBox	, id, (LTWH), Attr, Font, Grp, ‘Text’	Checkbox
FormPopUpTrigger	, id, (LTWH) , Attr, Font, Grp, ‘Text’	PopUp Trigger
FormSelectorTrigger	, id, (LTWH) , Attr, Font, Grp, ‘Text’	Selector trigger (ScrollBar)
FormRepeatButton	, id, (LTWH), , Attr, Font, Grp, ‘Text’	Repeating button
FormList	, id, (LTWH) , Attr, Font, (‘Text1’, ‘Text2’,…)	ListBox
FormGraffiti-StateIndicator	, id, (LT)	Graffiti® Shift Indicator
FormGadget	, id, Attr, (LTWH)	
FormBitmap	, id, Attr, (LT), Bitmap#	Form bitmap container. Bitmap# relates to a tBMP.

Example:

```
MainForm=(ResTFRM,1000,(0,0,160,160),0,0,MenuRes,
 (FormTitle,'DoodleHS'),
  Button1=(FormButton,, (3,145,36,12),,,,'Erase'),
);
(ResTVER,1000,'1.0');
```

11. Inline Assembler

Assembler procedures are used in conjunction with the inline assembler HSAsm. HSPascal has an extension to Pascal to provide inline assembly language via the ASM statement.

The general syntax is as follows:

```
AsmStatement=
  "asm" [Label:] AsmStmt {Separator AsmStmt} "end"
```

You can include assembly directly in your Pascal source files.

Example:

```
for n:=1 to 99 do begin
  ASM lea m,a0
    moveq #88,d1
    move.w d1,(a0)
  end;
  DoSomething;
  if Error then ASM trap #8 END; {Break into the debugger}
end;
```

HSPascal enables you to type assembly source directly into the Pascal source. The normal 68000 instruction set is supported.

The assembler does not know of the size of a used Pascal variable. It is up to you to specify the right size on each assembly instruction if not using the default word size of 16 bits. If you code `ASM move MyLongInt,D0 END;` you end up with only the hi part from the longint called MyLongInt.

These are the registers known by the assembler:

```
D0, D1, D2, D3, D4, D5, D6, D7,
A0, A1, A2, A3, A4, A5, A6, A7,
SP, USP, PC, CCR, SR
```

These are the 68000 instructions known by the assembler:

```
ABCD, ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ASL, ASR
BCC, BCHG, BCLR, BCS, BEQ, BGE, BGT, BHI, BHS, BLE, BLO, BLS, BLT,
BMI, BNE, BNZ, BPL, BRA, BSET, BSR, BTST, BVC, BVS, BZE
CHK, CLR, CMP, CMPA, CMPI, CMPM
DBCC, DBCS, DBEQ, DBF, DBGE, DBGT, DBHI, DBHS, DBLE, DBLO, DBLS, DBLT,
DBMI, DBNE, DBNZ, DBPL, DBRA, DBT, DBVC, DBVS, DBZE, DIVS, DIVU
EOR, EORI, EXG, EXT, ILLEGAL, JMP, JSR, LEA, LINK, LSL, LSR
MOVE, MOVEA, MOVEM, MOVEP, MOVEQ, MULS, MULU
NBCD, NEG, NEGX, NOP, NOT, OR, ORI, PEA
RESET, ROL, ROR, ROXL, ROXR, RTE, RTR, RTS
SBCD, SCC, SCS, SEQ, SF, SGE, SGT, SHI, SHS, SLE, SLO, SLS, SLT, SMI,
SNE, SNZ, SPL, ST, STOP, SUB, SUBA, SUBI, SUBQ, SUBX, SVC, SVS, SWAP,
SZE
TAS, TRAP, TRAPV, TST, UNLK
```

You will have to find the description of these instructions elsewhere outside this manual.

The assembler is accessed through an ASM statement. The syntax of an ASM statement is:
 ASM [Label:] AsmStmt < Separator AsmStmt > END;

where Label is an optional label and AsmStmt is one assembler statement. The Separator is either a semicolon or a new-line. Notice that this is different from the rest of the HSPascal compiler where a new-line only counts as a space. More than one AsmStmt can be placed on one line separated by semicolons or they can be placed one on each line just like that.

11.1 Labels

Labels declared in a Pascal label declaration can be used and/or defined in an ASM statement. But because assembly coding usually involves a lot of labels you can also make local labels in the ASM statement.

Local labels do not have to be declared in a label declaration but become known the first time they are used. They are declared by writing an at-sign (@) as the first character.

Example using labels
```
        BRA      @2
@1:     MOVE.B   (A1)+,(A0)+        //Move memory, count in D0
@2:     DBRA     D0,@1
```

11.2 ASSEMBLER procedures and functions

If the ASM END statement is the only statement in a procedure (or function) the procedure can be declared as an ASSEMBLER procedure. This removes the need for the BEGIN END block. The Add3 function can now be written as:

Example of a plain function using some assembler:
```
Function Add3(X, Y, Z: Integer): Integer;
begin
  ASM   move.w   X,d0               //8(a6)
        add.w    Y,d0               //10(a6)
        add.w    Z,d0               //12(a6)
        move.w   d0,14(a6)          // @result not possible yet!!
  END;
end;
```

Example of an assembler function:
```
Function Add3(X, Y, Z: Integer): Integer; ASSEMBLER;
ASM     move.w   X,d0               //8(a6)
        add.w    Y,d0               //10(a6)
        add.w    Z,d0               //12(a6)
        move.w   d0,14(a6)          // @result not possible yet!!
END;
```

It looks cleaner, but beside the obvious savings, the compiler also speeds up your code because of some optimization: Value parameters are never copied to local area. This saves code and time but you must be careful not to change these value variables.

In the following example the variable Src is a value parameter. As it can be seen in the appendix 'Inside HSPascal' section 'Value Parameters' only the address of a string is passed to the procedure. Because of the ASSEMBLER directive you only get the address of the original string. Without the ASSEMBLER directive you would have written LEA Src,A0 instead.

```
procedure CopyStr1(Src: String; var Dst: String); Assembler;
ASM     MOVE.L    Src,A0            //Get pointer
        MOVE.L    Dst,A1            //Get Pointer
@1:     MOVE.B   (A0)+,(A1)+
        BNE      @1
END;

procedure CopyStr2(Src: String; var Dst: String);
begin
ASM     LEA       Src,A0            //Get address, as Src string
        MOVE.L    Dst,A1            //has been copied to temp area
@1:     MOVE.B   (A0)+,(A1)+
        BNE      @1
END;
end;
```

11.4 Expressions

It is important to notice that expressions do not always act the same way in assembly as in Pascal.

Assume:

```
   Const
     X=4;
     Y=7;
   Var
     M,N,O: Integer;
     A: Array[0..99] of Integer;
```

We want to code the following in assembly:

```
   M:=X+Y;
   M:=N+X;
   M:=A[3];
```

The right way:

```
   ASM
     MOVE.W   #X+Y,M      {M:=X+Y}
     MOVE.W   N,D0        {M:=N+X}
     ADD.W    #Y,D0
     MOVE.W   D0,M
     MOVE.W   A+2*3,M     {M:=A[3]. 2 because Integer}
   END;
```

The wrong way:

```
   ASM
```

```
     MOVE.W   X+Y,M       {M:=X+Y}
     MOVE.W   N+X,M       {M:=N+X}
     MOVE.W   A,D0        {M:=A[3]}
     ADD.W    #3*2,D0
     MOVE.W   D0,M
   END;
```

11.5 Registers

These are the CPU registers known by the assembler:

Data registers:	*D0, D1, D2, D3, D4, D5, D6, D7,*
Data registers:	*A0, A1, A2, A3, A4, A5, A6, A7, SP*
Special registers:	*USP, PC, CCR, SR*

11.6 Addressing modes

The following is the 68000 CPU's addressing modes:

```
 Dn                Data register
 An                Address register direct
 (An)              Address register indirect
 (An)+             Address register indirect w/postincrement
 -(An)             Address register indirect w/predecrement
 d(An)             Address register indirect w/disp
 d(An,Rn)          Address register indirect w/disp and index
 XXXX              Absolute short
 d(PC)             Program counter indirect w/disp
 d(PC,Rn)          Program counter indirect w/disp and index
 #XXXX             Immediate data
```

The Rn index field can be Dn, An, Dn.S, An.S where S is W or L for word or long. The displacement d must be specified when using the two indexed formats. If displacement is zero then do write the zero.

11.7 Operators

The following are the defined operators for use in expressions:

Highest precedence:

+	Unary plus. Does nothing
-	Unary minus. Returns the negated value
!	Unary negation. Returns the bitwise not value

Second highest precedence:

*	Multiplication
/	Integer division

%	Integer remainder
&	Bitwise and
<<	Logical shift left
>>	Logical shift right

Lowest precedence:

+	Addition
-	Subtraction
|	Bitwise or
^	Bitwise xor

11.8 Limitations

Jumps with BRA, BSR etc can only be done using word offsets. BRA.B or BRA.S is not allowed.

The only way of getting an address in the code is by using LEA or PEA or by calling around it.

```
Procedure FrmSetEventHandlerNONE(PForm: FormPtr); Assembler;
ASM     jsr     @1                  //Call around
        moveq   #0,d0               //Simple "Eventhandler"
        rts
@1:     move.l  PForm,-(SP)
        SysTrap FrmSetEventHandler
end;
```

11.9 PalmOS, System calls

PalmOS are using C calling conventions for its functions and callback.

11.9.1. Functions calls:

```
//N:=StrLen(MyString)
        PEA     MyString     //MyString = Parameter 1
        SysTrap StrLen
        ADDQ    #4,SP        //Cleanup the stack
        MOVE.W  D0,N         //Use result
```

11.9.2. Callback (EventHandler, 1 parameter)

As a normal Pascal function has another calling convention, you need to manually handle this.
An EventHandler callback routine takes a pointer to a Event structure, and expects back a boolean.
The two type in question is (already defined in HSPascal):

```
Type
  EventPtr = ^ EventType;
  EventType = Record eType: xxx; xxxx end;
```

The parameter passed is a pointer to EventType and can be treated as either an EventPtr or more easily in Pascal as "Var Event: EventType". Try to not use pointers when you can avoid it.
The passed parameter must remain on the stack and as a Pascal function always do the

cleanup task, something special must be done here.
The return value of the function is to be returned in D0, but as Pascal returns this one on the stack, something special here too.

First we make the wanted function in 100% plain Pascal.

```
//Pascal version of what you want in your callback function
Function MainFormHandleEventPascal(var Event: EventType): Boolean;
Var
  PForm: FormPtr;
  Result: Boolean;
begin
  Result:=False;
  PForm := FrmGetActiveForm;
  with Event do
    case eType of
    frmOpenEvent:
      begin
        FrmDrawForm(PForm);
        .......
        Result:=True;
      end;
    penUpEvent:
      begin
        .......
        DrawCursor;
        Result:=True;
      end;
    end;
  MainFormHandleEventPascal:=Result
end;
```

Next we make the wrapper function.
We need to call MainFormHandleEventPascal using Pascal convention, that is

1. Clear a word on the stack for the result
2. Pass the first (and only) parameter
3. Call our function
4. Retrive the result and return in register D0

As the stack must be left as it was, we

1. need to always declare a procedure, not a function
   Just return the result in D0
2. cannot have parameters
   You must use 8(a6), 10(a6), 12(a6) and so on.

```
Procedure MainFormHandleEvent; Assembler;
ASM     clr.w   -(sp)                     //Room for result
        move.l  8(a6),-(sp)               //EventPtr
        bsr     MainFormHandleEventPascal //Call our real function
        move.b  (sp)+,d0                  //Get result into D0
end;
```

And install it the normal way, just remember to use the C-style version instead of the Pascal one.

```
FrmSetEventHandler(PForm,@MainFormHandleEvent);
```

11.9.3. Callback (more parameters)

When you need a callback with more parameters, remember the reverse nature of C-style parameter passing.

We want a callback to a C declaration as

```
void TableDrawItem(void *tableP, Int16 row, Int16 column, RectangleType
*bounds)
```

First we define a new Pascal-style function as

```
Procedure TableDrawItemPascal(TableP: Pointer; Row, Column: Int16; Var
Bounds: RectangleType);
```

Next we make the C to Pascal wrapper using assembler.

```
procedure TableDrawItem; Assembler;
ASM     //                              //No result required
        move.l  8(a6),-(sp)             //TableP
        move.w  12(a6),-(sp)            //Row
        move.w  14(a6),-(sp)            //Col
        move.l  16(a6),-(sp)            //Bounds
        bsr     TableDrawItemPascal     //Call our real function
        //                              //No result to D0
end;
```

12. Inside HSPascal

Some information needed for those who need a bit more specific information on the internal workings of HSPascal.

12.1 Memory Layout

Globals are located below A5. They are all initialized to zero value by Palm loader.

12.3 Data formats

12.3.1. Integer Types

Values are saved using as much space as needed to save all the information including a sign bit.

Integers, -32768..32767, are saved in a 16 bit word.

LongInts, -2147483648..2147483647, are saved in a 32 bit (long) word.

ShortInts, -128..127, are saved in an 8 bit word.

Bytes, 0..255, are saved in an 8 bit word.

Subranges of integers are saved using signed values. Subranges between the bounds of ShortInt, Integer or LongInt are stored using one of these formats. A subrange of 0..255 is therefore stored in a 16 bit word.

12.3.2. Char Types

The predefined type Char, #0..#255 is stored using an 8 bit word.

12.3.3. Enumerated Types

An enumerated type is stored as a byte if all items has an ordinality below 128, else it is saved using a 16 bit word.

12.3.4. Boolean Types

The Boolean type is an enumerated type of False..True and as such is saved in a byte.

12.3.5. String Types

The type **String** is allocated automatically. Internal they work like C strings, i.e. zero terminated. Normal string operations can be done like concatenations. Palm functions expecting a pointer to a C string can use a String right away without conversion. A string saved in a Pchar type (as a pointer) can be assigned to a String (S:=P). If the pointer is de-referenced, then only a char (first one) is used.

12.3.6. Real Types

The Single types are saved in 4 bytes using the IEEE standard format.

Singles will give you about 7 to 8 digits in the range of 3.4e-38..3.4e38

All HSPascal functions work using Single precision. MathLib is needed for doing double precision.

Do not use Reals for counting. In integer types, the LOWER xx bits are saved, always giving you access to add one in order to change the value. The real types only save the UPPER xx bits of the value, skipping the lesser significant bits.

Calculations on real types never make runtime errors.

12.3.7. Pointer Types

A Pointer is saved using a 32 bit longword. NIL is saved as a zero.

The Ptr function takes a LongInt and type converts it into a pointer. The 32 bit value returned is the same 32 bits as in the 32 bit LongInt.

12.3.8. Set Types

Set types are stored using 1 to 32 bytes. Each byte can hold eight elements. The lower and upper limits of the set are byte aligned before calculating the number of bytes needed.

A set of 6..10 uses 2 bytes because the limits are aligned to the range of 0..15.
The bit number within a byte is calculated as: Number = ElementNumber mod 8.

12.3.9. Array Types

Array elements are stored sequentially in memory with the element with the lowest index first in memory. A multi dimensional array is saved with the right-most dimension increasing first.

Example:
```
 Var
   A3: array[1..3,1..10,1..20] of integer.
```
A3[1,3] is then an array of [1..20] of integer.
A3[2] is then an array of [1..10,1..20] of integer.

12.3.10. Record Types

The first element of a Record is stored first in memory followed by the others in the same sequence as written. All variables using more than one byte are word aligned. All variant parts start at the same address.

12.4 Calling Conventions

Parameters for procedure (and function) calls are moved to the stack using the normal Pascal order for parameters. The first parameter met in the source is the first moved onto the stack. This is opposite to the C-language way of doing things. The called routine removes all parameters from the stack, except for the result from a function.

A procedure call looks like this:

```
MOVE     Param1,-(SP)
MOVE     Param2,-(SP)
MOVE     Param3,-(SP)
BSR      MyProc
```

A function call looks like this:

```
CLR      -(SP)
MOVE     Param1,-(SP)
MOVE     Param2,-(SP)
MOVE     Param3,-(SP)
BSR      MyFunc
MOVE     (SP)+,Result
```

The function call first makes room for the result on the stack, and then removes the result itself after the call.

If the result uses more than 4 bytes (like strings), then only the address is passed.

```
PEA      Result
MOVE     Param1,-(SP)
MOVE     Param2,-(SP)
MOVE     Param3,-(SP)
BSR      MyFunc
```

The procedure and function codes looks like this:

```
LINK     A6,#-StackUsed
MOVEM.L  D3-D7/A2-A4,-(SP)
{Take copy of value parameters}
....
MOVEM.L  (SP)+,D3-D7/A2-A4
UNLK     A6
MOVE.L   (SP)+,A0
ADD      #ParamSize,SP
JMP      (A0)
```

The last 3 lines may look like a single "`RTS`" if the routine takes no parameters. The "`MOVEM`" is removed if none of the registers d3-d7 or a2-a4 is used by the routine.

If the called procedure is nested (that is, if it is declared inside another procedure) the stack frame is also passed as a parameter (the last one). All parameters then appear to be four bytes higher up in memory from the A6 register.

A procedure call to a nested procedure looks like this:

```
MOVE     Param1,-(SP)
MOVE.L   A6,-(SP)
BSR      Procedure
```

The procedure can then use code as:

```
          MOVE.L    8(A6),A0
          MOVE      nn,-10(A0)
```

in order to read or write variables in other levels. If the nesting level is more than one, a number of "`MOVE.L 8(A0),A0`" will be inserted between these two lines.

12.4.1. Value Parameters

Parameters get copied every time the routine gets called. It the size of the variable is less than or equal to 4 bytes, its value is pushed onto the stack right away.

When the size is greater than 4 bytes a reference is pushed. A normal Pascal routine then makes its own copy as the first thing in the routine. An assembler routine does not have to take a copy if it does not change the parameter, thereby saving time and code.

Two and four byte parameters are passed using the same storage method as used when storing in normal variables.

Byte values are passed using the normal procedures for bytes on the stack, using a word. Otherwise the stack pointer would be misaligned.

Real types are passed as a 32 bit value.

Arrays and Records are passed by reference unless they can be contained in four bytes.

Strings are passed by reference.

Sets are passed as references to a maximized set of 256 elements, 32 bytes.

12.4.2. Variable Parameters

Variable parameters are always passed by pushing the address of the variable. The routine then uses the exact same location for its formal parameter as the caller does for its actual variable. Both variables must be exactly the same type.

12.4.3. Const Parameters

Const parameters are a parameter declaration preceded by **const** and followed by a type. Works like **var**, but some restrictions does apply, like you are not allowed to change the variable.

12.4.4. Function Results

For function results of Real, Integer, Char, Boolean and any subrange, two or four bytes are allocated on the stack before the call. The result is returned in this space and removed after the call.

For String types, the caller allocates temporary space for the string and pushes the address of this before any parameter.

Other types cannot be returned.

13. Compiler Directives

Compiler directives can be used for source code control.

```
 {$Define Debug}
 (*$Ifdef Debug *)
 {$R-}
 {xx$This is not a compiler directive! }
```

Compiler directives are commands not defined in standard Pascal. They are placed in special formed comments.
A directive must start with a $ dollar sign right after the starting bracket, as in '{$' or a '(*$'. This is followed by the command, which can be one or more letter(s).

13.1 Switch directives

Switch directives are written by using a single letter followed by a '+' or a '-'. A '+' enables the feature.

Several options can be given in one comment as in (*$D+,L-*).

13.1.1. Debug Information

{$D+} enables generation of debug information (MacsBug) in the generated program file. This makes the program size grow. Procedure and function names are those saved. A map file is also generated.

13.1.2. Line information, for debugger singlestepping

{$L+} enables output to the map file of line information, i.e. which sourceline makes which codeaddress.

13.1.3. Symbol table output

{$Y+} enables output to the map file of variables used, and their types.

13.2 Source code control

The compiler maintains some variables that are not part of the program being compiled. They can be used to control the way the compiler works, by enabling or disabling the compilation of certain parts of the source.

The following three variables are defined in this version: HSPascal, 68000 and Ver20. The last one is valid in this version release number 2.0.

The variables do not contain any value, you just test if they are there or not.

13.2.1. $Define, $Undef

New variables are defined with DEFINE, and removed by using the UNDEF directive.

The test for their presence is done with either IFDEF (for IF Defined THEN) or IFNDEF (for IF Not Defined THEN).

An ENDIF directive ends the $IFxx sequence.

The compiling state (between ifdef and endif) can be changed using the ELSE directive. Multiple ELSE is allowed, every occurrence will switch the state.

Finally you can test if a switch option is set by using the IFOPT directive.

```
 {$ifopt D}
```

13.2.2. $SetDefault

Save current settings. These settings will then be the default for all other files being compiled. Enables you to specify general options in the main source file and having the settings propagate into all files, i.e. like global settings on the command line.

14. Map file layout

14.1.1. Test program

```
Program MapFileTest;
{$D+,L+,Y+}  //Activate it all

Type
  TBox = Record Top,Left,Bottom,Right: Integer end;
var
  Box1,Box2: TBox;
  N: Integer;

procedure Test(var BoxA: TBox; const BoxB: TBox);
begin
  N:=3
end;

begin
  Test(Box1,Box2);
end.
```

14.1.2. Assembler code

```
0000: 6100 0050      BSR     0052     (call system init)
0004: 6600 000A      BNE     0010
0008: 6100 000A      BSR     0014     (call main program)
000C: 6100 00A0      BSR     00AE     (call system deinitialization)
0010: 7000           MOVEQ   #0,D0
0012: 4E75           RTS
0014: 4E56 0000      LINK    A6,#0    (main program)
0018: 486D FFCE      PEA     -50(A5)  (Box1)
001C: 486D FFC6      PEA     -58(A5)  (Box2)
0020: 6100 0016      BSR     0038     (call procedure Test)
0024: 4E5E           UNLK    A6
0026: 4E75 8C50 726F RTS              (MapFileTest as MacsBug name)
0038: 4E56 0000      LINK    A6,#0    (procedure Test)
003C: 3B7C 0003 FFC4 MOVE.W  #3,-60(A5) (N:=3)
0042: 4E5E           UNLK    A6
0044: 205F           MOVE.L  (A7)+,A0
0046: 504F           ADDQ.W  #8,A7
0048: 4ED0 8454 4553 JMP     (A0)      (TEST as MacsBug name)
0052: 4E56 FFF4      LINK    A6,#-12
0056: System initialization goes here
00AC: 4E75           RTS
00AE: System deinitialization goes here
00C4: 4E75           RTS
```

14.1.3. Map file generated

```
;HSPascal map file
;F File directory        F ; File# ; FileName
;P Procedure list        P ; ProcNo ; ProcName
;L Source lines:         L ; File# ; Line# ; Proc# ; RelCodeAddrHex
;A Procedure addresses   A ; Proc# ; AbsStartHex ; LenHex
;V Variable              V ; Name ; Offset ; ProcNo ; Level ;
                             TypCh? ; Type# ; Rec#
;T Type                  T ; T# ; Name ; Typ ; Size ; Base ;
                             Min ; Max ; Element ; Index ; Rec#
;----
```

```
T;2;Longint;Int;4;2;-2147483648;2147483647;;;
T;1;Integer;Int;2;2;-32768;32767;;;
V;Bottom;4;0;-1;R;1;1
V;Left;2;0;-1;R;1;1
V;Right;6;0;-1;R;1;1
V;Top;0;0;-1;R;1;1
T;54;TBox;Rec;8;0;0;0;;;1   (Type #54=TBox=Record, size=8)
V;Box2;-58;0;0;G;54;  (Var Box2 is at -58(A5) , type=#54=TBox)
V;Box1;-50;0;0;G;54;  (Var Box1 is at -58(A5) , type=#54=TBox)
V;N;-60;0;0;G;1;      (Var N is at -60(A5))
;----
V;BoxB;8;70;1;H;54;
V;BoxA;12;70;1;H;54;
L;1;11;70;0000
L;1;12;70;0004
L;1;13;70;0004
L;1;15;71;0000
L;1;16;71;0004
L;1;17;71;0010
F;1;X:\Somewhere\MapFileTest.pas
F;2;X:\Somewhere\Units\HSSys1.pas
F;3;X:\Somewhere\Units\HSSys2.pas
F;4;X:\Somewhere\Units\HSSys3.pas
F;5;X:\Somewhere\Units\HSSys4.pas
F;6;X:\Somewhere\Units\HSSys5.pas
F;7;X:\Somewhere\Units\HSUTILS.pas
F;8;X:\Somewhere\Units\System\FLOATMGR.pas
F;9;X:\Somewhere\Units\System\FEATUREMGR.pas
F;10;X:\Somewhere\Units\TRAPS31.pas
F;11;X:\Somewhere\Units\TRAPS30.pas
F;12;X:\Somewhere\Units\System\ERRORBASE.pas
F;13;X:\Somewhere\Units\TRAPS33.pas
F;14;X:\Somewhere\Units\System\SYSTEMMGR.pas
F;15;X:\Somewhere\Units\TRAPS35.pas
F;16;X:\Somewhere\Units\UI\MENU.pas
F;17;X:\Somewhere\Units\System\WINDOW.pas
F;18;X:\Somewhere\Units\System\FONT.pas
F;19;X:\Somewhere\Units\System\RECT.pas
F;20;X:\Somewhere\Units\System\BITMAP.pas
F;21;X:\Somewhere\Units\System\SYSTEMRESOURCES.pas
F;22;X:\Somewhere\Units\System\DATAMGR.pas
F;23;X:\Somewhere\Units\System\MEMORYMGR.pas
P;70;MAPFILETEST.TEST
P;71;MAPFILETEST.ProgramBegin
A;70;0038;001A
A;71;0014;0024
```

15. Command Line Compiler

The HSPascal compiler also has a command line version called HSPC. It takes all of the command from the command line and from a configuration file "HSPC.CFG".
Call it as:

```
   HSPC {Option} FileName {Option}
```

The call "HSPC -$D- Clock" will make HSPC compile the file Clock.Pas with no debugging information.

The call "HSPC" alone will write a help screen.

Each option starts with a "/" or a "-", both are accepted.

These are the options:

```
  -$D+   Debug information in mapfile
  -$L+   Linedebug information in mapfile

  -V     Verbose (show more)

  -Dxxx  Define conditionals
  -Oxxx  PRG & UNI directory
  -Ixxx  Include directories
```

15.1 $ Switch Option

The -$ or /$ option allow you to enable or disable compiler options, just as if they are entered into the first line of your program.

Example:

```
 HSPC -$D-  -$L-  DEMO
```

Look in Chapter 13, Compiler Directives for a description of all compiler directive options.

15.2 Configuration File

The file HSPC.CFG contains initial compiler definitions that you do not want to enter each time you start HSPC. It is simply a number of options entered line by line. These options get read before the command line gets parsed, allowing you to define your own default settings.

16. Compiler and Runtime Errors

16.1 Compiler errors

The most general error given by the compiler is when something is expected.

These are the common simple errors:

';' expected
':' expected
',' expected
'(' expected
')' expected
'[' expected
']' expected
'.' expected
'=' expected
':=' expected
'..' expected
END expected
DO expected
OF expected
THEN expected
TO or DOWNTO expected

BEGIN expected
The compiler expects BEGIN when it cannot find any further declarations in a function.

INTERFACE expected
After UNIT aName; the only thing allowed is Uses AllMyUnits; followed by the word INTERFACE.

IMPLEMENTATION expected
The compiler expects IMPLEMENTATION when it cannot find any further declarations.

The following are also common errors, but more complex than the previous.

Constant expected
Boolean expression expected
The construct "IF expression THEN" needs a boolean expression. You cannot write IF 2+3 THEN.

Integer constant expected
The real assignment R:=123; is ok, but the assignment R:=12345678; is bad because the compiler first reads the number as an integer. You can add a '.0' to make it a real.

Integer expression expected
Integer variable expected
Integer or real constant expected
Integer or real expression expected
Integer or real variable expected
Pointer variable expected
Typed Pointer variable expected
Record variable expected
Ordinal type expected
Ordinal expression expected
String constant expected
String expression expected
String variable expected
Identifier expected
Variable expected
Type identifier expected
Field identifier expected
Char expression expected

The following are different types of errors:

Unknown identifier
Undefined type in pointer definition

'.' expected after module name
Duplicate identifier

Type mismatch
Constant out of range
Constant and CASE types do not match
Operand types does not match operator
Invalid result type
A function only returns simple types including string and real types.

Invalid string length
Invalid subrange base type
Lower bound greater than upper bound
Invalid FOR control variable
The FOR variable must either be declared in same function or global. It must not be part of any structure.

Illegal assignment
String constant exceeds line
You may have forgotten an apostrophe in the string. Note that a single apostrophe in a string must be written as two apostrophes.

Error in integer constant
Error in real constant
Division by zero
Structure too large

Constants are not allowed here
Invalid type cast argument
Invalid '@' argument
Only works on functions, procedures and variables.

Invalid GOTO
Label not within current block
Undefined label
Label already defined

Set base type out of range

Error in type
Error in statement
Error in expression

Undefined FORWARD procedure: xx

Too many symbols
There is a fixed number of Kbytes set aside for the symbol table for each unit or program.

Too many nested scopes
Also given when too many units are used.

Expression too complicated
All data registers in the 68000 processor are used for temporaries.

Too many variables
Too much code

Bad unit or program name: xx
Unit missing: xx
Unit not found
Duplicate unit name: xx

Syntax error
Something is wrong!

Unexpected end of text
You may have more begins than ends. Maybe caused by an non-terminated comment.

Line too long
Only 250 characters are allowed on one line.

No room in 16 bit fix-up field
Current max size of HSPascal generated programs is 32 Kbyte.

Conditional variable missing
Misplaced conditional directive
ENDIF directive missing

17. HSPascal versus other Pascal Compilers

There will always be some differences, some minor and some major between different compiler versions.

HSPascal has no file io routines. But Write(S) and Writeln(S) are implemented in Crt unit.

The American National Standard (ANS) Institute has defined the standard for Pascal. HSPascal will be compared against this one and Turbo Pascal. The comparison is not complete as far as the minor details are concerned.

17.1 HS Pascal versus Turbo Pascal for the PC

One important thing: HSPascal is running on a 68000 micro processor, while Turbo Pascal is running on a 80x86 processor. Both compilers uses the CPU's default way of saving data in memory.

The 68000 does not allow reads and writes at odd addresses if the element size is not a byte. Therefore all variables using more space than a byte are allocated on even addresses. Furthermore the 68000 saves words (and others) in a non-reversed order in memory, whereas Turbo Pascal uses the '86 reversed mode.

In Turbo Pascal you can stop a FOR loop by assigning the ending value to the control variable. This does not work in HSPascal. In ANS Pascal this is not allowed.

The use of NIL pointers is not checked in Turbo Pascal. In HSPascal nil pointers will be trapped by the operating system (Fatal error dialog). Likewise the use of uninitialized pointers may sometimes stop the program.

17.2 HSPascal versus ANS Pascal

HSPascal has no file io routines.

HSPascal cannot tell you if any non-standard features are used.

The Put and Get system in ANS Pascal is not implemented.

Procedures and functions do not take Procedure and Function parameters in HSPascal.

The parameter to New and Dispose cannot specify which variant part to use. You may use GetMem and FreeMem instead.

Pack and Unpack are not implemented.

The @ symbol is an operator in HSPascal. The same as the ^ in ANS Pascal.

17.3 Extensions to ANS Pascal

HSPascal cannot tell you if any non-standard features are used.

HSPascal implements strings with dynamic length, and routines to operate on them.

HSPascal implements the types ShortInt, LongInt, Single and other Palm-specific types.

HSPascal implements the unit technique of modular programming.

Labels can be both identifiers and the normal digit sequence.

Integer constants can be written in hexadecimal format when preceded with a $.

All character values can be written in character and string constants when preceded with a # or a ^. A decimal or hex value can be written after the #.

The sequence of Label, Const, Var, Type, Procedure and Function declarations can be in any order.

Three new logical operators: xor, shl and shr. The logical operators also work as bitwise operators.

Enumeration can take explicitly assigned ordinalities, as in C# and Delphi6. See Chapter 3.3.4.

The @ operator works like the Addr function. In ANS Pascal @ and ^ is the same.

HSPascal implements value and variable typecasts.

An ELSE statement is allowed as the default case in a CASE construction. There is no default in ANS Pascal.

A variable formal parameter in a procedure/function can be typeless.

The list of reserved words can be seen in the list in chapter 2.4, Reserved Words.

18. Pascal for C programmers

Most programs for the Palm you will come across are programmed using the C programming language. The operating system itself is built around C. As an example, text strings are implemented as zero-terminated strings.

We have done a lot to ensure that it is at least as easy to program in HSPascal as in C. Strings are implemented as zero-terminated ones, as the OS expects it this way. However, HSPascal has gone further in making it easy. Strings normally do not have to be allocated before use as they are declared as String[nn], where the space is already allocated. And, of course, strings can be concatenated by using the plus sign, as is the normal way to work in Pascal.

This chapter is meant to help you in translating or understanding a C program, so that it can be built in the same way using Pascal.

Normally you should not translate everything literally. C uses a lot of pointers. In Pascal you would often work with passing the data around to a procedure by using the VAR prefix. This is in effect the same, but requires less thought when using such a procedure, and makes the program look more straightforward.

18.1 Constants

C	Pascal	Type
123	123	decimal
123L	123	decimal
0x123	$123	hex
O123	-	octal
'A'	'A'	character
"asd"	'asd'	string

18.2 Types

The basic types used in the Palm are implemented as simple translation from e.g. UInt16 to Integer etc.
Pascal has a Boolean type, that should be used when the meaning is true/false (yes/no). Otherwise an integer would be called for, using the values of 0 and 1.

18.2.1. Enumerated values

C	Pascal
enum TColor { Red, Green, Blue };	TColor = (Red, Green, Blue);

In C a variable of type TColor can be assigned to an integer. In Pascal, the types do not match, so a typecast is needed: AnInteger:=Ord(MyColor).

18.2.2. Arrays

Arrays in C are zero based. You do not specify the lower bounds as it is always 0. Only the element count is specified, as in

C	Pascal
int ZipCodes[1000];	ZipCodes: Array[0..999] of Integer;

18.2.3. Pointers

C	Pascal	
void pointer	Pointer	Declaration
UInt16 *Zip	Zip: ^UInt16;	Declaration
*Link	Link^	Use (dereference)
&Data	@Data	Use (take address)
NULL	NIL	'empty' pointer

18.2.4. Structures / Records

C

```
 struct customer {
   struct customer  *next;
   char name[50+1];
   char address[50+1];
   int vip;
   union
     char whynot [20+1];
   union
    char  VipID [20+1]
 }
```

Pascal

```
 Type
   PCustomer = ^TCustomer;
   TCustomer = Record
     Next: PCustomer;
     Name, Address: String[50];
     Case vip: Boolean of
     False: (WhyNot: String[20]);
     True:  (VipID: String[20]);
   end;
```

18.3 Procedures / Functions

19. FAQ

19.1 How to make a new program icon from a Windows bitmap

Save the picture in a suitable format, i.e. 1, 2, 4 or 8 bits color (1 is black and white)
In PilRC use the BITMAP FAMILY for resource TAIN,1000 as in:

```
ICONFAMILY "HSPc1.bmp" "" "HSPc4.bmp" "HSPc8.bmp"
```

in HSPascal include the file as `{$R TAIN03e8.bin}`

19.2 How to include a Windows bitmap into a program

In PilRC use the BITMAP as:

```
BITMAP bitmapClock3  "Images/Clock03.bmp" COMPRESS
```

in HSPascal include the file as {$R Tbmpxxxx.bin}

19.3 How to load a Bitmap (Tbmp #2000)

```
var
  bmpH: MemHandle;
  Bitmap: BitmapPtr;
  Err: Integer;
begin
  bmpH := DmGetResource( s2u32('Tbmp'), 2000);
  if bmpH=NIL then fail!!
  BitMap := BitMapPtr(MemHandleLock(bmpH));
  WinDrawBitMap(BitMap, X,Y);
  Err := MemHandleUnlock(bmpH);
  Err := DmReleaseResource(bmpH);
```

19.4 Draw a string value ('Hello World') into a screenfield (id2000)

Make two procedures like

```
Function S2Handle(Const S: String): MemHandle;
var
  NewH: MemHandle;
  NewP: ^String;
  Err:  Integer;
begin
  NewH := MemHandleNew(Length(S)+1);
  NewP := MemHandleLock(NewH);
  NewP^ := S;
  Err:=MemHandleUnlock(NewH);
  S2Handle := NewH;
end;

Procedure S2Field(FldID: UInt16; Const S: String);
var
  Fld: FieldPtr;
```

```
    OldH, NewH: MemHandle;
    Err: Integer;
  begin
    Fld := FrmGetObjectPtr(FrmGetActiveForm, FldID);
    OldH := FldGetTextHandle(Fld);
    NewH := S2Handle(S);
    FldSetTextHandle(Fld, NewH);
    if OldH<>NIL then Err := MemHandleFree(OldH);
    FldDrawField(Fld)
  end;
```

Then call

```
    S2Field(2000, 'Hello World');
```

19.5 Draw an integer value (123) into a screenfield (id2000)

```
  Uses HSUtils;
  begin
    S2Field(2000, i2s(123));
  end;
```

19.6 How to select items from a StringList

```
  // Will not walk past an empty string.
  Procedure StringByIndex(var S: PChar; Index: Integer);
  var N: Integer; P: ^Longint;
  begin
    P:=@S;
    for N:=2 to Index do begin
      if S^=#0 then EXIT;   // Stop on empty string, like in #0 #0
      Inc(P^, Length(S)+1); // Inc S
    end;
  end;
```

20. Booklist, literature