Amsterdam Compiler Kit-Pascal reference manual

by

Johan W. Stevenson

(January 4, 1983)

(revised)

Hans van Eck

(May 1, 1989)

Vakgroep Informatica
Vrije Universiteit
De Boelelaan 1081
Amsterdam

1. Introduction

This document refers to the (1982) BSI standard for Pascal [1]. Ack-Pascal complies with the requirements of level 1 of BS 6192: 1982, with the exceptions as listed in this document.

The standard requires an accompanying document describing the implementation-defined and implementation-dependent features, the reaction on errors and the extensions to standard Pascal. These four items will be treated in the rest of this document, each in a separate chapter. The other chapters describe the deviations from the standard and the list of options recognized by the compiler.

The Ack-Pascal compiler produces code for an EM machine as defined in [2]. It is up to the implementor of the EM machine to decide whether errors like integer overflow, undefined operand and range bound error are recognized or not.

There does not (yet) exist a hardware EM machine. Therefore, EM programs must be interpreted, or translated into instructions for a target machine. The Ack-Pascal compiler is currently available for use with the VAX, Motorola MC68020, Motorola MC68000, PDP-11, and Intel 8086 code-generators. For the 8086, MC68000, and MC68020, floating point emulation is used. This is made available with the *-fp* option, which must be passed to *ack*[3].

2. Implementation-defined features

For each implementation-defined feature mentioned in the BSI standard we give the section number, the quotation from that section and the definition. First we quote the definition of implementation-defined:

> Possibly differing between processors, but defined for any particular processor.

BS 6.1.7: Each string-character shall denote an implementation-defined value of the required char-type.

> All 7-bits ASCII characters except linefeed LF (10) are allowed.

BS 6.4.2.2: The values of type real shall be an implementation-defined subset of the real numbers denoted as specified by 6.1.5 bu signed real.

> The format of reals is not defined in EM. Even the size of reals depends on the EM-implementation. The compiler can be instructed, by the V-option, to use a different size for real values. The size of reals is preset by the calling program *ack* [3] to the proper size.

BS 6.4.2.2: The type char shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations.

> The 7-bits ASCII character set is used, where LF (10) denotes the end-of-line marker on text-files.

BS 6.4.2.2: The ordinal numbers of the character values shall be values of integer-type, that are implementation-defined, and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero.

> The normal ASCII ordering is used: ord('0')=48, ord('A')=65, ord('a')=97, etc.

BS 6.6.5.2: The post-assertions imply corresponding activities on the external entities, if any, to which the file-variables are bound. These activities, and the point at which they are actually performed, shall be implementation-defined.

> The reading and writing writing of objects on files is buffered. This means that when a program terminates abnormally, IO may be unfinished. Terminal IO is unbuffered. Files are closed whenever they are rewritten or reset, or on program termination.

BS 6.7.2.2: The predefined constant maxint shall be of integer-type and shall denote an implementation-defined value, that satisfies the following conditions:

(a) All integral values in the closed interval from -maxint to +maxint shall be values of the integer-type.

(b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.

(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.

(d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

The representation of integers in EM is a *n**8-bit word using two's complement arithmetic. Where *n* is called wordsize. The range of available integers depends on the EM implementation: For 2-byte machines, the integers range from -32767 to +32767. For 4-byte machines, the integers range from -2147483647 to 2147483647. The number -maxint-1 may be used to indicate 'undefined'.

BS 6.7.2.2: The result of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined

Since EM doesn't specify floating point format, it is not possible to specify the accuracy. When the floating point emulation is used, and the default size of reals is 8 bytes, the accuracy is 11 bits for the exponent, and 53 bits for the mantissa. This gives an accuracy of about 16 digits, and exponents ranging from -309 to +307.

BS 6.9.3.1: The default TotalWidth values for integer, Boolean and real types shall be implementation-defined.

The defaults are:

integer	6 for 2-byte machines, 11 for 4-byte machines
Boolean	5
real	14

BS 6.9.3.4.1: ExpDigits, the number of digits written in an exponent part of a real, shall be implementation-defined.

ExpDigits is defined as 3. This is sufficient for all implementations currently available. When the representation would need more than 3 digits, then the string '***' replaces the exponent.

BS 6.9.3.4.1: The character written as part of the representation of a real to indicate the beginning of the exponent part shall be implementation-defined, either 'E' or 'e'.

The exponent part starts with 'e'.

BS 6.9.3.5: The case of the characters written as representation of the Boolean values shall be implementation-defined.

The representations of true and false are 'true' and 'false'.

BS 6.9.5: The effect caused by the standard procedure page on a text file shall be implementation-defined.

The ASCII character form feed FF (12) is written.

BS 6.10: The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-defined if the variable is of a file-type.

The program parameters must be files and all, except input and output, must be declared as such in the program block.

The program parameters input and output, if specified, will correspond with the UNIX streams 'standard input' and 'standard output'.

The other program parameters will be mapped to the argument strings provided by the caller of this program. The argument strings are supposed to be path names of the files to be opened or created. The order of the program parameters determines the mapping: the first parameter is mapped onto the first argument string etc. Note that input and output are ignored in this mapping.

The mapping is recalculated each time a program parameter is opened for reading or writing by a call to the standard procedures reset or rewrite. This gives the programmer the opportunity to manipulate the list of string arguments using the external procedures argc, argv and argshift available in libpc [6].

BS 6.10: The effect of an explicit use of reset or rewrite on the standard textfiles input or output shall be implementation-defined.

The procedures reset and rewrite are no-ops if applied to input or output.

3. Implementation-dependent features

For each implementation-dependent feature mentioned in the BSI standard, we give the section number, the quotation from that section and the way this feature is treated by the Ack-Pascal system. First we quote the definition of 'implementation-dependent':

> Possibly differing between processors and not necessarily defined for any particular processor.

BS 6.7.2.1: The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

Operands are always evaluated, so the program part

```
if (p<>nil) and (p^.value<>0) then
```

is probably incorrect.

The left-hand operand of a dyadic operator is almost always evaluated before the right-hand side. Some peculiar evaluations exist for the following cases:

1. the modulo operation is performed by a library routine to check for negative values of the right operand.

2. the expression

   ```
   set1 <= set2
   ```

   where set1 and set2 are compatible set types is evaluated in the following steps:

   - evaluate set2
   - evaluate set1
   - compute set2+set1
   - test set2 and set2+set1 for equality

3. the expression

   ```
   set1 >= set2
   ```

   where set1 and set2 are compatible set types is evaluated in the following steps:

   - evaluate set1
   - evaluate set2
   - compute set1+set2
   - test set1 and set1+set2 for equality

BS 6.7.3: The order of evaluation, accessing and binding of the actual-parameters for functions shall be implementation-dependent.

> The order of evaluation is from right to left.

BS 6.8.2.2: The decision as to the order of accessing the variable and evaluating the expression in an assignment-statement, shall be implementation-dependent.

> The expression is evaluated first.

BS 6.8.2.3: The order of evaluation and binding of the actual-parameters for procedures shall be implementation-dependent.

> The same as for functions.

BS 6.9.5: The effect of inspecting a text file to which the page procedure was applied during generation is implementation-dependent.

> The formfeed character written by page is treated like a normal character, with ordinal value 12.

BS 6.10: The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-dependent unless the variable is of a file-type.

> Only variables of a file-type are allowed as program parameters.

4. Error handling

There are three classes of errors to be distinguished. In the first class are the error messages generated by the compiler. The second class consists of the occasional errors generated by the other programs involved in the compilation process. Errors of the third class are the errors as defined in the standard by:

> An error is a violation by a program of the requirements of this standard that a processor is permitted to leave undetected.

4.1. Compiler errors

Error are written on the standard error output. Each line has the form:
<file>, line <number>: <description>
Every time the compiler detects an error that does not have influence on the code produced by the compiler or on the syntax decisions, a warning messages is given. If only warnings are generated, compilation proceeds and probably results in a correctly compiled program.

Sometimes the compiler produces several errors for the same line. They are only shown up to a maximum of 5 errors per line. Warning are also shown up to a maximum of 5 per line.

Extensive treatment of these errors is outside the scope of this manual.

4.2. Runtime errors

Errors detected at run time cause an error message to be generated on the diagnostic output stream (UNIX file descriptor 2). The message consists of the name of the program followed by a message describing the error, possibly followed by the source line number. Unless the -L-option is turned on, the compiler generates code to keep track of which source line causes which EM instructions to be generated. It depends on the EM implementation whether these LIN instructions are skipped or executed.

For each error mentioned in the standard we give the section number, the quotation from that section and the way it is processed by the Pascal-compiler or runtime system.

For detected errors the corresponding message and trap number are given. Trap numbers are useful for exception-handling routines. Normally, each error causes the program to terminate. By using exception-handling routines one can ignore errors or perform alternate actions. Only some of the errors can be ignored by restarting the failing instruction. These errors are marked as non-fatal, all others as fatal. A list of errors with trap number between 0 and 63 (EM errors) can be found in [2]. Errors with trap number between 64 and 127 (Pascal errors) are listed in [7].

BS 6.4.6: It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by T1.

The compiler distinguishes between array-index expressions and the other places where assignment-compatibility is required.

Array subscripting is done using the EM array instructions. These instructions have three arguments: the array base address, the index and the address of the array descriptor. An array descriptor describes one dimension by three values: the lower bound on the index, the number of elements minus one and the element-size. It depends on the EM implementation whether these bounds are checked. Since most implementations don't, an extra compiler flag is added to force these checks.
The other places where assignment-compatibility is required are:

- assignment
- value parameters
- procedures read and readln
- the final value of the for-statement

For these places the compiler generates an EM range check instruction, except when the R-option is turned on, or when the range of values of T2 is enclosed in the range of T1. If the expression consists of a single variable and if that variable is of a subrange type, then the subrange type itself is taken as T2, not its host-type. Therefore, a range instruction is only generated if T1 is a subrange type and if the expression is a constant, an expression with two or more operands, or a single variable with a type not enclosed in T1. If a constant is assigned, then the EM optimizer removes the range check instruction, except when the value is out of bounds.

It depends on the EM implementation whether the range check instruction is executed or skipped.

BS 6.4.6: It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

This error is not detected.

BS 6.5.3.3: It shall be an error if a component of a variant-part of a variant, where the selector of the variant-part is not a field, is accessed unless the variant is active for the entirety of each reference and access to each component of the variant.

This error is not detected.

BS 6.5.4: It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined.

The EM definition does not specify the binary representation of pointer values, so that it is not possible to choose an otherwise illegal binary representation for the pointer value NIL. Rather arbitrary the compiler uses the integer value zero to represent NIL. For all current implementations this does not cause problems.

The size of pointers depends on the implementation and is preset in the compiler by *ack* [3]. The compiler can be instructed, by the V-option, to use another size for pointer objects. NIL is represented here by the appropriate number of zero words.

It depends on the EM implementation whether de-referencing of a pointer with value NIL causes an error.

BS 6.5.4: It shall be an error to remove the identifying-value of an identified variable from its pointer-type when a reference to the variable exists.

When the identified variable is an element of the record-variable-list of a with_statement, a warning is given at compile-time. Otherwise, this error is not detected.

BS 6.5.5: It shall be an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

When f is altered when it is an element of the record-variable-list of a with-statement, a warning is given. When a buffer-variable is used as a variable-parameter, an error is given. This is done at compile-time.

BS 6.6.5.2: It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the file handling procedures rewrite, put, reset and get.

For each of these four operations the pre-assertions can be reformulated as:

rewrite(f):	no pre-assertion.
put(f):	f is opened for writing and f^ is not undefined.
reset(f):	f exists.
get(f):	f is opened for reading and eof(f) is false.

The following errors are detected for these operations:

rewrite(f):

more args expected, trap 64, fatal:
f is a program-parameter and the corresponding file name is not supplied by the caller of the program.

rewrite error, trap 101, fatal:
the caller of the program lacks the necessary access rights to create the file in the file system or operating system problems like table overflow prevent creation of the file.

put(f):

file not yet open, trap 72, fatal:
reset or rewrite are never applied to the file. The checks performed by the run time system are not foolproof.

not writable, trap 96, fatal:
f is opened for reading.

write error, trap 104, fatal:
probably caused by file system problems. For instance, the file storage is exhausted. Because IO is buffered to improve performance, it might happen that this error occurs if the file is closed. Files are closed whenever they are rewritten or reset, or on program termination.

reset(f):

more args expected, trap 64, fatal:
: same as for rewrite(f).

reset error, trap 100, fatal:
: f does not exist, or the caller has insufficient access rights, or operating system tables are exhausted.

get(f):

file not yet open, trap 72, fatal:
: as for put(f).

not readable, trap 97, fatal:
: f is opened for writing.

end of file, trap 98, fatal:
: eof(f) is true just before the call to get(f).

read error, trap 103, fatal:
: unlikely to happen. Probably caused by hardware problems or by errors elsewhere in the program that destroyed the file information maintained by the run time system.

truncated, trap 99, fatal:
: the file is not properly formed by an integer number of file elements. For instance, the size of a file of integer is odd.

non-ASCII char read, trap 106, non-fatal:
: the character value of the next character-type file element is out of range (0..127). Only for text files.

BS 6.6.5.3: It shall be an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

This error is not detected.

BS 6.6.5.3: It shall be an error to use dispose(q) if the identifying variable has been allocated using the form new(p,c1,...,cn).

This error is not detected. However, this error can cause more memory to be freed then was allocated. Dispose causes a fatal trap 73 when memory already on the free list is freed again.

BS 6.6.5.3: It shall be an error to use dispose(q,k1,...,km) if the identifying variable has been allocated using the form new(p,c1,...,cn) and m is not equal to n.

This error is not detected. However, this error can cause more memory to be freed then was allocated. Dispose causes a fatal trap 73 when memory already on the free list is freed again.

BS 6.6.5.3: It shall be an error if the variants of a variable to be disposed are different from those specified by the case-constants to dispose.

This error is not detected.

BS 6.6.5.3: It shall be an error if the value of the pointer parameter of dispose has nil-value or is undefined.

The same comments apply as for de-referencing NIL or undefined pointers.

BS 6.6.5.3: It shall be an error if a variable created using the second form of new is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

This error is not detected.

BS 6.6.6.2: It shall be an error if the value of sqr(x) does not exist.

This error is detected for real-type arguments (real overflow, trap 4, non-fatal).

BS 6.6.6.2: It shall be an error if x in ln(x) is smaller than or equal to 0.

This error is detected (error in ln, trap 66, non-fatal)

BS 6.6.6.2: It shall be an error if x in sqrt(x) is smaller than 0.

This error is detected (error in sqrt, trap 67, non-fatal)

In addition to these errors, overflow in the expression exp(x) is detected (error in exp, trap 65, non-fatal; real overflow, trap 4, non-fatal)

BS 6.6.6.3: It shall be an error if the integer value of trunc(x) does not exist.

It depends on the implementations whether this error is detected. The floating-point emulation detects this error (conversion error, trap 10, non-fatal).

BS 6.6.6.3: It shall be an error if the integer value of round(x) does not exist.

It depends on the implementations whether this error is detected. The floating-point emulation detects this error (conversion error, trap 10, non-fatal).

BS 6.6.6.4: It shall be an error if the integer value of ord(x) does not exist.

This error can not occur, because the compiler will not allow such ordinal types.

BS 6.6.6.4: It shall be an error if the character value of chr(x) does not exist.

Except when the R-option is off, the compiler generates an EM range check instruction. The effect of this instruction depends on the EM implementation.

BS 6.6.6.4: It shall be an error if the value of succ(x) does not exist.

Same comments as for chr(x).

BS 6.6.6.4: It shall be an error if the value of pred(x) does not exist.

Same comments as for chr(x).

BS 6.6.6.5: It shall be an error if f in eof(f) is undefined.

This error is detected (file not yet open, trap 72, fatal).

BS 6.6.6.5: It shall be an error if f in eoln(f) is undefined, or if eof(f) is true at that time.

> The following errors may occur:
>
> > file not yet open, trap 72, fatal;
> > not readable, trap 97, fatal;
> > end of file, trap 98, fatal.

BS 6.7.1: It shall be an error if a variable-access used as an operand in an expression is undefined at the time of its use.

> The compiler performs some limited checks to see if identifiers are used before they are set. Since it can not always be sure (one could, for instance, jump out of a loop), only a warning is generated. When an expression contains a function-call, an error occur if the function is not assigned at runtime.

BS 6.7.2.2: A term of the form x/y shall be an error if y is zero.

> It depends on the EM implementation whether this error is detected. On some machines, a trap may occur.

BS 6.7.2.2: It shall be an error if j is zero in 'i div j'.

> It depends on the EM implementation whether this error is detected. On some machines, a trap may occur.

BS 6.7.2.2: It shall be an error if j is zero or negative in i MOD j.

> This error is detected (only positive j in 'i mod j', trap 71, non-fatal).

BS 6.7.2.2: It shall be an error if the result of any operation on integer operands is not performed according to the mathematical rules for integer arithmetic.

> The reaction depends on the EM implementation. Most implementations, however, will not notice integer overflow.

BS 6.8.3.5: It shall be an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

> This error is detected (case error, trap 20, fatal).

BS 6.9.1: It shall be an error if the sequence of characters read looking for an integer does not form a signed-integer as specified in 6.1.5.

> This error is detected (digit expected, trap 105, non-fatal).

BS 6.9.1: It shall be an error if the sequence of characters read looking for a real does not form a signed-number as specified in 6.1.5.

> This error is detected (digit expected, trap 105, non-fatal).

BS 6.9.1: When read is applied to f, it shall be an error if the buffer-variable f^ is undefined or the pre-assertions for get do not hold.

> This error is detected (see get(f)).

BS 6.9.3: When write is applied to a textfile f, it shall be an error if f is undefined or f is opened for reading.

> This error is detected (see put(f)). Furthermore, this error is also detected when f is not a textfile.

BS 6.9.3.1: The values of TotalWidth or FracDigits shall be greater than or equal to one; it shall be an error if either value is less then one.

> When either value is less than zero, an error (illegal field width, trap 75, non-fatal) occurs. Zero values are allowed, in order to maintain some compatibility with the old Ack-Pascal compiler.

BS 6.9.5: It shall be an error if the pre-assertion required for writeln(f) doe not hold prior to the invocation of page(f);

> This error is detected (see put(f)).

5. Extensions to the standard

1. External routines

Except for the required directive ’forward’ the Ack-Pascal compiler recognizes the directive ’extern’. This directive tells the compiler that the procedure block of this procedure will not be present in the current program. The code for the body of this procedure must be included at a later stage of the compilation process.

This feature allows one to build libraries containing often used routines. These routines do not have to be included in all the programs using them. Maintenance is much simpler if there is only one library module to be changed instead of many Pascal programs.

Another advantage is that these library modules may be written in a different language, for instance C or the EM assembly language. This is useful for accessing some specific EM instructions not generated by the Pascal compiler. Examples are the system call routines and some floating point conversion routines. Another motive could be the optimization of some time-critical program parts.

The use of external routines, however, is dangerous. The compiler normally checks for the correct number and type of parameters when a procedure is called and for the result type of functions. If an external routine is called these checks are not sufficient, because the compiler can not check whether the procedure heading of the external routine as given in the Pascal program matches the actual routine implementation. It should be the loader’s task to check this. However, the current loaders are not that smart. Another solution is to check at run time, at least the number of words for parameters. Some EM implementations check this.

For those who wish the use the interface between C and Pascal we give an incomplete list of corresponding formal parameters in C and Pascal.

```
Pascal                C
a:integer             int a
a:char                int a
a:boolean             int a
a:real                double a
a:^type               type *a
var a:type            type *a
procedure a(pars)     struct {
                          void (*a)() ;
                          char *static_link ;
                      }
function a(pars):type struct {
                          type (*a)() ;
                          char *static_link ;
                      }
```

The Pascal runtime system uses the following algorithm when calling function/procedures passed as parameters.

```
if ( static_link )      (*a)(static_link,pars) ;
else                    (*a)(pars) ;
```

2. Separate compilation.

The compiler is able to (separately) compile a collection of declarations, procedures and functions to form a library. The library may be linked with the main program, compiled later. The syntax of these modules is

```
module = [constant-definition-part]
        [type-definition-part]
        [var-declaration-part]
        [procedure-and-function-declaration-part]
```

The compiler accepts a program or a module:

```
unit = program | module
```

All variables declared outside a module must be imported by parameters, even the files input and output. Access to a variable declared in a module is only possible using the procedures and functions declared in that same module. By giving the correct procedure/function heading followed by the directive 'extern' procedures and functions declared in other units may be used.

3. Assertions.

When the s-option is off, Ack-Pascal compiler recognizes an additional statement, the assertion. Assertions can be used as an aid in debugging and documentation. The syntax is:

```
assertion = 'assert' Boolean-expression
```

An assertion is a simple-statement, so

```
simple-statement = [assignment-statement |
                    procedure-statement |
                    goto-statement |
                    assertion
                   ]
```

An assertion causes an error if the Boolean-expression is false. That is its only purpose. It does not change any of the variables, at least it should not. Therefore, do not use functions with side-effects in the Boolean-expression. If the a-option is turned on, then assertions are skipped by the compiler. 'assert' is not a word-symbol (keyword) and may be used as identifier. However, assignment to a variable and calling of a procedure with that name will be impossible. If the s-option is turned on, the compiler will not know a thing about assertions, so using assertions will then give a parse error.

4. Additional procedures.

Three additional standard procedures are available:

halt: a call of this procedure is equivalent to jumping to the end of the program. It is always the last statement executed.

The exit status of the program may be supplied as optional argument. If not, it will be zero.

release:
mark: for most applications it is sufficient to use the heap as second stack. Mark and release are suited for this type of use, more suited than dispose. mark(p), with p of type pointer, stores the current value of the heap pointer in p. release(p), with p initialized by a call of mark(p), restores the heap pointer to its old value. All the heap objects, created by calls of new between the call of mark and the call of release, are removed and the space they used can be reallocated. Never use mark and release together with dispose!

5. UNIX interfacing.

If the c-option is turned on, then some special features are available to simplify an interface with the UNIX environment. First of all, the compiler allows for a different type of string constants. These string constants are delimited by double quotes ('"'). To put a double quote into these strings, the double quote must be repeated, like the single quote in normal string constants. These special string constants are terminated by a zero byte (chr(0)). The type of these constants is a pointer to a packed array of characters, with lower bound 1 and unknown upper bound.
Secondly, the compiler predefines a new type identifier 'string' denoting this just described string type.

These features are only useful for declaration of constants and variables of type 'string'. String objects may not be allocated on the heap and string pointers may not be de-referenced. Still these strings are very useful in combination with external routines. The procedure write is extended to print these zero-terminated strings correctly.

6. Double length (32 bit) integers.

If the d-option is turned on, then the additional type 'long' is known to the compiler. By default, long variables have integer values in the range -2147483647..+2147483647, but this can be changed with the -V option (if the backend can support this). Long constants can not be declared. Longs can not be used as control-variables. It is not allowed to form subranges of type long. All operations allowed on integers are also allowed on longs and are indicated by the same operators: '+', '-', '*', '/', 'div', 'mod'. The procedures read and write have been extended to handle long arguments correctly. It is possible to read longs from a file of integers and vice-versa, but only if longs and integers have the same size. The default width for longs is 11. The standard procedures 'abs' and 'sqr' have been extended to work on long arguments. Conversion from integer to long, long to real, real to long and long to integer are automatic, like the conversion from integer to real. These conversions may cause a

conversion error, trap 10, non-fatal

7. Underscore as letter.

The character '_' may be used in forming identifiers, if the u- or U-option is turned on. It is forbidden to start identifiers with underscores, since this may cause name-clashes with run-time routines.

8. Zero field width in write.

Zero TotalWidth arguments are allowed. No characters are written for character, string or Boolean type arguments then. A zero FracDigits argument for fixed-point representation of reals causes the fraction and the character '.' to be suppressed.

9. Pre-processing.

If the very first character of a file containing a Pascal program is the sharp ('#', ASCII 23(hex)) the file is preprocessed in the same way as C programs. Lines beginning with a '#' are taken as preprocessor command lines and not fed to the Pascal compiler proper. C style comments, /*......*/, are removed by the C preprocessor, thus C comments inside Pascal programs are also removed when they are fed through the preprocessor.

6. Deviations from the standard

Ack-Pascal deviates from the standard proposal in the following ways:

1. Standard procedures and functions are not allowed as parameters in Ack-Pascal. The same result can be obtained with negligible loss of performance by declaring some user routines like:

```
function sine(x:real):real;
begin
   sine:=sin(x)
end;
```

2. The standard procedures read, readln, write and writeln are implemented as word-symbols, and can therefore not be redeclared.

7. Compiler options

Some options of the compiler may be controlled by using "{$....}". Each option consists of a lower case letter followed by +, - or an unsigned number. Options are separated by commas. The following options exist:

a +/- this option switches assertions on and off. If this option is on, then code is included to test these assertions at run time. Default +.

c +/- this option, if on, allows the use of C-type string constants surrounded by double quotes. Moreover, a new type identifier 'string' is predefined. Default -.

d +/- this option, if on, allows the use of variables of type 'long'. Default -.

i <num> with this flag the setsize for a set of integers can be manipulated. The number must be the number of bits per set. The default value is wordsize-1.

l +/- if + then code is inserted to keep track of the source line number. When this flag is switched on and off, an incorrect line number may appear if the error occurs in a part of the program for which this flag is off. These same line numbers are used for the profile, flow and count options of the EM interpreter em [5]. Default +.

r +/- if + then code is inserted to check subrange variables against lower and upper subrange limits. Default +.

s +/- if + then the compiler will hunt for places in the program where non-standard features are used, and for each place found it will generate a warning. Default -.

t +/- if + then each time a procedure is entered, the routine 'procentry' is called, and each time a procedure exits, the procedure 'procexit' is called. Both 'procentry' and 'procexit' have a 'string' as parameter. This means that when a user specifies his or her own procedures, the c-option must be used. Default procedures are present in the run time library. Default -.

u +/- if + then the character '_' is treated like a letter, so that it may be used in identifiers. Procedure and function identifiers are not allowed to start with an underscore because they may collide with library routine names. Default -.

Some of these flags (c, d, i, s, u, C and U) are only effective when they appear before the 'program' symbol. The others may be switched on and off.

A very powerful debugging tool is the knowledge that inaccessible statements and useless tests are removed by the EM optimizer. For instance, a statement like:

```
if debug then
  writeln('initialization done');
```

is completely removed by the optimizer if debug is a constant with value false. The first line is removed if debug is a constant with value true. Of course, if debug is a variable nothing can be removed.

A disadvantage of Pascal, the lack of preinitialized data, can be diminished by making use of the possibilities of the EM optimizer. For instance, initializing an array of reserved words is sometimes optimized into 3 EM instructions. To maximize this effect variables must be initialized as much as possible in order of declaration and array entries in order of decreasing index.

8. References

[1] BSI standard BS 6192: 1982 (ISO 7185).

[2] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, E.G.Keizer, "Description of a machine architecture for use with block structured languages", Informatica rapport IR-81.

[3] UNIX manual ack(I).

[4] UNIX manual ld(I).

[5] UNIX manual em(I).

[6] UNIX manual libpc(VII)

[7] UNIX manual pc_prlib(VII)