

NAKED MINI® 4

PASCAL EXTENSIONS

ADDENDUM

93170-80

NOVEMBER 1981



NAKED MINI® Division

Engineering Reference Number
90-93170-01A0

HERTFORD HOUSE
DENHAM WAY, MAPLE CROSS
RICKMANSWORTH, HERTFORDSHIRE WD3 7XD

Telephone Rickmansworth 71211 Telex 922654

Printed in U.S.A.

REVISION HISTORY

<u>Revision</u>	<u>Issue Date</u>	<u>Comments</u>
A0	Nov. 1981	Original issue.

Computer Automation, Inc., reserves the right to make changes in specifications at any time and without notice. The information furnished by Computer Automation, Inc., in this publication is believed to be accurate and reliable. However, no responsibility is assumed by Computer Automation, Inc., for its use; nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of Computer Automation, Inc.

Computer Automation, Inc., welcomes comments and suggestions for improvements to our manuals. Should difficulty be encountered with the information in the manual or if you have a suggestion for manual improvement, please send the comments or suggestions to Computer Automation, Inc., Technical Publications, Department 11-0770, 18651 Von Karman, Irvine, California, 92713. Please provide as much information as possible to help evaluate the comment or suggestion.

PREFACE

The objective of this Addendum is to describe ISO Standard PASCAL extensions provided by NAKED MINI® 4 PASCAL.

In order to provide the NAKED MINI® 4 PASCAL programmer with additional information, Appendix A presents the PASCAL Syntax List, and Appendices B and C provide summary descriptions of implementation-defined features and implementation-dependent features, as called for in Clause 6 of ISO Standard PASCAL.

The user is also referred to Pascal Programming Structure (Reston, VA: Reston Pub. Co; 1980), by George W. Cherry, for general information on the PASCAL language.

CONTENTS

SECTION 1	NAKED MINI 4 PASCAL SYNTAX EXTENSION.....	1-1
	1.0 OVERVIEW.....	1-1
	1.1 IDENTIFIERS.....	1-1
	1.2 DIRECTIVE.....	1-1
	1.3 STATEMENTS.....	1-1
SECTION 2	SEPARATE COMPILEATION EXTENSIONS.....	2-1
	2.0 OVERVIEW.....	2-1
	2.1 EXTERNAL DIRECTIVE.....	2-1
	2.2 SEPARATE COMPILEATION.....	2-1
	2.3 SHARING VARIABLES.....	2-2
SECTION 3	IDENTIFIERS.....	3-1
	3.0 OVERVIEW.....	3-1
	3.1 TYPE EXTENSIONS.....	3-1
	3.1.1 Long.....	3-1
	3.1.2 String.....	3-1
	3.2 PROCEDURE EXTENSIONS.....	3-2
	3.2.1 Halt.....	3-2
	3.2.2 Mark and Release.....	3-2
	3.3 FUNCTION EXTENSION.....	3-2
SECTION 4	STATEMENTS.....	4-1
	4.0 OVERVIEW.....	4-1
	4.1 ASSERT STATEMENT.....	4-1
	4.2 THE OTHERWISE EXTENSION TO THE CASE STATEMENT.....	4-1

APPENDIXES

A	PASCAL SYNTAX LIST.....	A-1
B	NAKED MINI 4 PASCAL IMPLEMENTATION-DEFINED FEATURES.....	B-1
C	NAKED MINI 4 PASCAL IMPLEMENTATION-DEPENDENT FEATURES.....	C-1

SECTION 1

NAKED MINI® 4 PASCAL SYNTAX EXTENSIONS

1.0 OVERVIEW

This section briefly summarizes extensions implemented by NAKED MINI® 4 PASCAL. Subsequent sections describe each extension in further detail.

1.1 IDENTIFIERS

Extended identifier types, procedures and functions are as follows:

<u>Type</u>	<u>Procedure</u>	<u>Function</u>
LONG	HALT RELEASE MARK	TYPE-AS-FUNCTION

1.2 DIRECTIVE

One additional directive, **extern**, is implemented by NAKED MINI® 4 PASCAL.

1.3 STATEMENTS

An **assert** statement is implemented by NAKED MINI® 4 PASCAL.

An **otherwise:** label extension is implemented for the case statement.

SECTION 2

SEPARATE COMPILATION EXTENSION

2.0 OVERVIEW

This section describes the extensions permitting separate compilation of procedures and sharing of variables between the separately compiled procedures.

2.1 EXTERNAL DIRECTIVE

The directive **extern** or **external** tells the compiler that the block portion of this procedure will not be present in the current program. The body of the procedure can be supplied via a separate compilation, and combined during the link process. This feature allows the separate compilation of sets of procedures and functions, permitting the assembler object code to be merged to form a library.

Library modules can also be written in NAKED MINI® 4 assembly language. This can be useful for optimizing time-critical program parts that use specific instructions not generated by the PASCAL compiler.

2.2 SEPARATE COMPILATION

The use of **extern** (or **external**) enables the building of libraries containing often used routines. This enables the compiler to separately compile a collection of declarations, procedures, and functions to form a library which can later be linked with the main program.

An example of a procedure calling an external function follows:

```
function Factorial (X:integer); extern;
procedure Main;
  var A,B:integer;
  begin
    A:= Factorial (B)
  end;

function Factorial (X:integer);
  begin
    Factorial :=X
    while X>1 do
      X:=X-1;
      Factorial:= Factorial*X;
    end;
  end;
```

2.3 SHARING VARIABLES

All the variables declared above the procedure level (not in a procedure) are allocated in a segment which the linker treats as **global common**. Therefore, separately compiled procedures may share the same variables provided that the corresponding variables are declared in the same order in each procedure (e.g., integer variables A,B declared in procedure Main correspond with declared variables aa,bb in "external" procedure Secondary). Note that the linker does not test for variable type compatibility.

The PASCAL compiler places all variables in a segment named **globvec:** in the same order they appeared during compilation. Interfacing assembly language programs can therefore access these variables in this segment.

SECTION 3

IDENTIFIER

3.0 OVERVIEW

Extended type, procedure and function identifiers provided by NAKED MINI® 4 PASCAL are described in this section.

3.1 TYPE EXTENSIONS

There are two **type** extensions: **long** and **string**.

3.1.1 Long

To take advantage of the NAKED MINI® 4 computer capabilities, NM4 PASCAL adds the data type **long** to the standard set. This new data type is a 32-bit signed integer. Type **long** variables have integer values in the range -2147483648 to +2147483647; **long** constants can be declared, but the user cannot 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 standard procedures **read(ln)** and **write(ln)** have been extended to handle **long** arguments correctly. The default width for **long** is 11 characters. The standard functions **abs** and **sqr** support **long** arguments. Conversion from **integer** to **long**, **long** to **real**, **real** to **long** and **long** to **integer** is automatic, like the conversion from **integer** to **real**.

3.1.2 String

If the **c**-option is turned on, a special feature is provided to simplify interfacing with the operating system. This feature allows the compiler to accept **string** constants of a different type. These **string** constants are delimited by double quotes ("). To place a double quote within one of these strings, the double quote must be repeated once (") analogous to the way a single quote would be placed in a regular string ('). 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 a lower bound of 1 and an unknown upper bound. After the **string** type is described, the compiler predefines a new type identifier "**string**."

This option permits constants and variables of type **string** to be declared only. It does not permit **string** objects to be allocated on the heap or **string** pointers to be de-referenced. These **strings** are very useful in combination with external routines. The procedure **writeln** is extended to print these zero-terminated strings correctly.

3.2 PROCEDURE EXTENSIONS

Three additional standard procedures are provided with NAKED MINI[®] 4 PASCAL: **Halt**, **Mark**, and **Release**. These are discussed in the following paragraphs.

3.2.1 Halt

A call to 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 can be supplied as an optional argument.

3.2.2 Mark and Release

In NM4 PASCAL, all pointer-referenced data is cumulatively allocated storage by one or more executions of **new**. The result is a second stack of pointer-referenced data called the "heap." **Mark** and **Release** (rather than **dispose**) are tailored for handling this data.

Mark (P), with p of type pointer stores the current value of the heap pointer in p. **Release** (P) (with p initialized by the **mark** (P) call) restores the heap pointer to its old value. All 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.

3.3 FUNCTION EXTENSION

The function extension allows the user to bypass type checking as called for in ISO Standard PASCAL, by means of the type-as-function syntax.

When it is desired to mix variables of different types (standard simple types) in expressions or assignment statements, a construct of the form

intrinsic-or-user-defined-TYPE (expression)

causes that construct to be treated as if it were of type

intrinsic-or-user-defined-TYPE

No conversion operations are performed, but this construct permits the internal form of a specific type to be treated as if it were of another size-compatible type. If the type of the expression is not size-compatible with the intrinsic-or-user-defined-TYPE, a syntax error is produced.

SECTION 4

STATEMENTS

4.0 OVERVIEW

A new statement, **assert**, and an extension to the **case** statement are described in this section.

4.1 ASSERT STATEMENT

The NAKED MINI® 4 PASCAL compiler recognizes an additional statement, **assert**, which is provided as an aid in debugging and documentation. The syntax is:

```
<assert> ::= <'assert' Boolean-expression>
```

The purpose of an **assert** is to produce an error whenever the associated Boolean-expression is false. It does not change any variables. Since the **assert** statement is compiled under optional control, it should not be used with functions that have side effects.

The **assert** statement is not a key word and can thus be used as an identifier. However, assignment to a variable and calling of a procedure with that name is not allowed.

4.2 THE OTHERWISE EXTENSION TO THE CASE STATEMENT

The label **otherwise:** has been added as an option within the **case** statement. It permits a convenient escape for items that do not match the conditions specified by the other labels within a **case** statement. In the following example,

```
case    <expression> of
        <1st case label>:    <statement 1>;
        <2nd case label>    <statement 2>;
        ...
        otherwise:          <statement no match>;
end
```

all expressions that do not match the specified list of labels would execute <statement no match>.

PASCAL SYNTAX LIST

Syntax Symbols

: a variable name and type	; statement separator
:= assignment operator	.. subrange specifier
alternatively	[x] 0 or 1 instance of x
= relational operator	{x} 0 or more instance of x
↑ file and pointer variable indicator	< > syntactic construct
. terminator	

[illegible]

<file identifier> ::= <identifier>

```
<identifier> ::= <letter> { <letter or digit>}
```

```
<letter or digit> ::= <letter> | <digit>
```

```
<block> ::= <label declaration part> <constant definition part>
           <type definition part> <variable declaration part>
           <procedure and function declaration part>
           <statement part>
```

[illegible]

```
<label> ::= <unsigned integer>
```

```
<constant definition part> ::= <empty> |  
    const <constant definition> { ; <constant definition> } ;
```

```
<constant definition> ::= <identifier> = <constant>
```

```
<constant> ::= <unsigned number> | <sign> <unsigned number> |  
               <constant identifier> | <sign> <constant identifier> |  
               <string>
```

```
<unsigned number> ::= <unsigned integer> | <unsigned real>
```

```
<unsigned integer> ::= <digit> {<digit>}
```

```

<unsigned real> ::= <unsigned integer> , <digit> {<digit>} |
    <unsigned integer> , <digit> (<digit>)* E <scale factor>
    <unsigned integer> E <scale factor>

```

<scale factor> ::= <unsigned integer> | <sign> <unsigned integer>

<sign> ::= + | -

<constant identifier> ::= <identifier>

<string> ::= '<character> {<character>}'

<type definition part> ::= <empty> |
 type <type definition> { ; <type definition> } ;

<type definition part> ::= <identifier> = <type>

<type> ::= <simple type> | <structured type> | <pointer type>

<simple type> ::= <scalar type> | <subrange type> |
 <type identifier>

<scalar type> ::= (<identifier> { , <identifier> })

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<structured type> ::= <unpacked structured type> |
 packed <unpacked structured type>

<unpacked structured type> ::= <array type> | <record type> |
 <set type> | <file type>

<array type> ::= array [<index type> { , <index type> }] of
 <component type>

<index type> ::= <simple type>

<component type> ::= <type>

<record type> ::= record <field list> end

<field list> ::= <fixed part> | <fixed part> ; <variant part> |
 <variant part>

<fixed part> ::= <record section> { ; <record section> }

<record section> ::= <field identifier> { , <field identifier> } :
 <type> | <empty>

<variant part> ::= case <tag field> <type identifier> of
 <variant> { ; <variant> }

<tag field> ::= <field identifier> ; | <empty>

<variant> ::= <case label list> ; (<field list>) | <empty>

```

<case label list> ::= <case label> { , <case label> }

<case label> ::= <constant> | otherwise:

<set type> ::= set of <base type>

<base type> ::= <simple type>

<file type> ::= file of <type>

<pointer type> ::= ↑<type identifier>

<variable declaration part> ::= <empty> |
    var <variable declaration> { ; <variable declaration> } ;

<variable declaration> ::= <identifier> { , <identifier> } ; <type>

<procedure and function declaration part> ::= <empty> |
    {<procedure or function declaration> ;}

<procedure or function declaration> ::= <procedure declaration> |
    <function declaration>

<procedure declaration> ::= <procedure heading> <block>

<procedure heading> ::= procedure <identifier> ; |
    procedure <identifier> ( <formal parameter section>
        { ; <formal parameter section> } ) ;

<formal parameter section> ::= <parameter group> |
    var <parameter group> | function <parameter group> |
    procedure <identifier> { , <identifier> }

<parameter group> ::= <identifier> { , <identifier> } :
    <type identifier>

<function declaration> ::= <function heading> <block>

<function heading> ::= function <identifier> ; <result type> ; |
    function <identifier> ( <formal parameter section>
        { ; <formal parameter section> } ) ; <result type> ;

<result type> ::= <type identifier>

<statement part> ::= <compound statement>

<statement> ::= <unlabeled statement> |
    <label> : <unlabeled statement>

<unlabeled statement> ::= <simple statement> |
    <structured statement>

```

```

<simple statement> ::= <assignment statement> |
                    <procedure statement> | <go to statement> |
                    <empty statement>

<assignment statement> ::= <variable> := <expression> |
                           <function identifier> := <expression>

<variable> ::= <entire variable> | <component variable> |
              <referenced variable>

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable> ::= <indexed variable> | <field designator> |
                        <file buffer>

<indexed variable> ::= <array variable> [ <expression>
                                         { , <expression> } ]

<array variable> ::= <variable>

<field designator> ::= <record variable> , <field identifier>

<record variable> ::= <variable>

<field identifier> ::= <identifier>

<file buffer> ::= <file variable> ↑

<file variable> ::= <variable>

<referenced variable> ::= <pointer variable> ↑

<pointer variable> ::= <variable>

<expression> ::= <simple expression> | <simple expression>
                <relational operator> <simple expression>

<relational operator> ::= = | <> | < | <= | >= | > | in

<simple expression> ::= <term> | <sign> <term> |
                      <simple expression> <adding operator> <term>

<adding operator> ::= + | - | or

<term> ::= <factor> | <term> <multiplying operator> <factor>

<multiplying operator> ::= * | / | div | mod | and

<factor> ::= <variable> | <unsigned constant> | ( <expression> ) |
            <function designator> | <set> | not <factor>

```

<unsigned constant> ::= <unsigned number> | <string> |
 <constant identifier> | nil

<function designator> ::= <function identifier> |
 <function identifier> (<actual parameter>
 { , <actual parameter> })

<function identifier> ::= <identifier>

<set> ::= [<element list>]

<element list> ::= <element> { , <element> } | <empty>

<element> ::= <expression> | <expression> .. <expression>

<procedure statement> ::= <procedure identifier> |
 <procedure identifier> (<actual parameter>
 { , <actual parameter> })

<procedure identifier> ::= <identifier>

<actual parameter> ::= <expression> | <variable> |
 <procedure identifier> | <function identifier>

<go to statement> ::= goto <label>

<empty statement> ::= <empty>

<empty> ::=

<structured statement> ::= <compound statement> |
 <conditional statement> | <repetitive statement> |
 <with statement>

<compound statement> ::= begin <statement> { : <statement> } end

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= if <expression> then <statement> |
 if <expression> then <statement> else <statement>

<case statement> ::= case <expression> of <case list element>
 { ; <case list element> } end

<case list element> ::= <case label list> : <statement>
 <empty>

<case label list> ::= <case label> { , <case label> }

<repetitive statement> ::= <while statement> | <repeat statement> |
 <for statement>

<while statement> ::= while <expression> do <statement>

<repeat statement> ::= repeat <statement> { ; <statement> }
 until <expression>

<for statement> ::= for <control variable> := <for list> do
 <statement>

<for list> ::= <initial value> to <final value> |
 <initial value> downto <final value>

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<with statement> ::= with <record variable list> do <statement>

<record variable list> ::= <record variable> { , <record variable> }

APPENDIX B

NAKED MINI[®] 4 PASCAL IMPLEMENTATION-DEFINED FEATURES

Implementation-defined

"Parts of language which may differ between processors, but which will be defined for any particular processor" (CA NM4).

ISO 6.1.7

Each ASCII character (except linefeed LF (10)) denotes a value of character type. An apostrophe must be doubled within a string.

ISO 6.4.2.2

The values of type **real**:

Base for exponent part	2
Precision of fraction	24
Minimum exponent	-127
Maximum exponent	+127

ISO 6.4.2.2

ASCII character set is used where LF (10) denotes end-of-line marker on text files.

ISO 6.4.2.2

ASCII order of character values (consecutive non-negative integer values starting at zero):

$\text{ord}("0") = 48, \text{ord}("A") = 65, \text{ord}("a") = 97, \text{etc.}$

ISO 6.4.3.4

The largest and smallest values of **integer**-type permitted as numbers of a value of **set**-type. The smallest value is 0. The largest value is default 15, but can be changed by using the **i**-option of the compiler up to a maximum of 32767. The compiler allocates as many bits for **set**-type variables as are necessary to store all possible values of the host-type of the base-type of the set, rounded up to the nearest multiple of 16. If 8 bits are sufficient then only 8 bits are used if part of a packed structure. Thus, the variable **s**, declared by

```
var s: set of "0".."9";
```

will contain 128 bits, not 10 or 16. These 128 bits are stored in 16 bytes, both for packed and unpacked sets. If the host-type of the base-type is integer, then the number of bits depends on the i-option. The programmer may specify how many bits to allocate for these sets. The default is 16, the maximum is 32767. The effective number of bits is rounded up to the next multiple of 16, or up to 8 if the number of bits is less than or equal to 8. Note that the use of set-constructors for sets with more than 256 elements is far less efficient than for smaller sets.

ISO 6.7.2.2

Predefined constant Maxint is of integer type. Maxint = 32767. The range of available integers is -32768 through 32767.

ISO 6.9.4.2

Default total width default values:

Integer	6
Boolean	5
Real	13
Long	11

ISO 6.9.4.5.1

ExpDigits, the number of digits written in the exponent part of a real: 2

ISO 6.9.4.5.1

Character written as part of a real to indicate beginning of the exp part: "e".

ISO 6.9.4.6

Representation of true and false: "true", "false".

ISO 6.9.6

The effect on a text file (caused by standard procedure page): ASCII formfeed character FF(12) is written.

ISO 6.10

Program parameters (relative to external entities) must be files which must be declared in program block.

ISO 6.10

Reset, rewrite procedures are no-ops if applied to input or output.

APPENDIX C

NAKED MINI[®] 4 PASCAL IMPLEMENTATION-DEPENDENT FEATURES

Implementation-dependent

"Parts of language which may differ between processors, and for which there need not be a definition for a particular processor" (CA NM4).

ISO 6.1.4

Directive, external. NAKED MINI[®] 4 PASCAL compiler will recognize one directive, **extern** (or **external**), in addition to standard directive **forward**.

ISO 6.7.2.1

Operand Evaluation Sequence. 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:

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

This is the only case where the right-hand side is computed first.

3. The expression

set1 >= set2

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

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

ISO 6.7.3

Parameter Evaluation, Function.

The order of evaluation and binding is from left to right.

ISO 6.8.2.2

Expression Evaluation Sequence.

If access to the variable involves the indexing of an array and/or a reference to a field within a variant of a record and/or the de-referencing of a pointer-variable and/or a reference to a buffer-variable, the decision whether these actions precede or follow the evaluation of the expression shall be implementation-dependent.

Expression is evaluated first.

ISO 6.8.2.3

Parameter, Order of Evaluation, and Binding Procedure.

The order of evaluation and binding is from left to right.

ISO 6.9.6

Formfeed Character.

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

ISO 6.10

External Variable.

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