Aalborg Universitetscenter
Institut for Elektroniske Systemer
Laboratoriet
Strandvejen 19
9000 Ålborg

2559

dde

MIKADOS

Pascal User´s Guide

Dansk Data Elektronik ApS
4 December 1979

Authors: Rolf Molich
Mette Staugård

Copyright 1979
Dansk Data Elektronik ApS

Table of contents

1. Introduction 1.1

2. Getting started 2.1

3. System overview 3.1

4. Using the system 4.1
- 4.1 Running the compiler 4.2
- 4.2 Compile time options 4.6
- 4.3 Passing parameters to a Pascal program 4.10
- 4.4 Generating a new standard interpreter 4.11

5. System intrinsics 5.1
- 5.1 String handling 5.2
  - 5.1.1 Length of string 5.2
  - 5.1.2 Locate pattern in string (POS) 5.3
  - 5.1.3 Concatenate strings 5.3
  - 5.1.4 Copy string 5.4
- 5.2 Input/output 5.5
  - 5.2.1 Open/create file 5.6
  - 5.2.2 Close file 5.8
  - 5.2.3 Check for end-of-file/end-of-line 5.8
  - 5.2.4 Determine result of i/o operation 5.9
  - 5.2.5 Read/write record from direct access file 5.9
  - 5.2.6 Read/write record from sequential file 5.10
  - 5.2.7 Eject page (PAGE) 5.10
  - 5.2.8 Position to record in direct access file (SEEK) 5.10
  - 5.2.9 Edit string on output device 5.11
  - 5.2.10 Clear screen on output device 5.13
- 5.3 Character array manipulation 5.14
  - 5.3.1 Scan character array 5.14

5.3.2 Move character array (MOVELEFT,MOVERIGHT) 5.15
5.3.3 Initialize character array (FILLCHAR) 5.16
5.4 Miscellaneous useful routines 5.17
5.4.1 Determine size of structure (SIZEOF) 5.17
5.4.2 Read system clock (TIME) 5.17
5.4.3 Compute power of ten (PWROFTEN) 5.18
5.4.4 Dynamic memory control (MARK, RELEASE) 5.18
5.4.5 Move cursor (GOTOXY) 5.18
5.4.6 Basic input/output operations (IN80, OUT80) 5.19
5.4.7 Set i/o result 5.20
5.4.8 Delay program 5.20
5.4.9 Start new program (CHAIN) 5.21

6. Segment procedures and functions 6.1

7. Differences between MIKADOS Pascal and standard Pascal 7.1
7.1 Character set and special symbols 7.1
7.2 CASE statements 7.2
7.3 Dynamic memory allocation 7.3
7.4 GOTO and EXIT statements 7.5
7.5 Packed variables 7.6
7.5.1 Packed arrays 7.6
7.5.2 Packed records 7.9
7.5.3 Using packed variables as parameters 7.11
7.5.4 PACK and UNPACK standard procedures 7.12
7.6 Parametric procedures and functions 7.12
7.7 Program headings 7.12
7.8 The standard types 7.13
7.8.1 Integer 7.13
7.8.2 Real 7.13
7.9 Sets 7.14
7.10 Strings 7.15
7.11 Extended comparisons 7.18

7.12 READ and READLN	7.18
7.13 WRITE and WRITELN	7.19
7.14 PUT and GET	7.21
7.15 EOF - end-of-file	7.21
7.16 EOLN - end-of-line	7.23
7.17 Miscellaneous implementation size limits	7.23
8. External procedures and functions	8.1
8.1 Transfer of parameters	8.1
8.2 Parameter formats	8.3
8.3 Linking external procedures	8.5
9. Pascal E (Extended precision Pascal)	9.1
9.1 Interpreter and compiler names	9.1
9.2 Intrinsic functions	9.2
9.3 Names of interpreter modules	9.2
9.4 Direct support of READREAL and WRITEREAL	9.3
9.5 Internal representation of real numbers	9.3
Appendix A. Compile time error messages	A.1
Appendix B. Run time (interpreter) error messages	B.1
Appendix C. Revised syntax diagrams	C.1
Appendix D. System performance data	D.1
D.1 Space requirements	D.1
D.2 Execution times	D.2
Appendix E. Summary of manual changes	E.1

1. Introduction

This manual describes the MIKADOS Pascal compiler and interpreter system running under the multiprogrammed MIKADOS operating system on the ID-7000 and SPC/1 microcomputers designed and manufactured by Dansk Data Elektronik ApS (DDE).

The MIKADOS Pascal compiler accepts source programs written in the standard Pascal high level computer language (with minor restrictions and extensions) and translates them into code for a hypothetical computer known as the "P-machine". This code may subsequently be executed by using an interpreter program, which simulates the hypothetical computer on an ID-7000 or SPC/1 computer running under the MIKADOS operating system.

This manual is a reference manual describing the differences between MIKADOS Pascal and standard Pascal as defined in

> Kathleen Jensen and Niklaus Wirth
> PASCAL: User Manual and Report
> Second Edition
> Springer Verlag New York Inc. 1975 (or 1978)

Throughout this manual many references will be made to the above Pascal standard reference guide (referred to as "J&W").

The reader of this manual is expected to be familiar with the Pascal language. This manual is not a tutorial book on MIKADOS Pascal.

The MIKADOS Pascal system is a modified version of the Pascal compiler and INTEL 8080 interpreter constructed by a team at the University of California, San Diego, (UCSD), directed by professor Kenneth Bowles. Users of the Pascal system should note that any malfunction of the system is the sole responsibility of Dansk Data Elektronik ApS.

Portions of this manual have been adapted from the UCSD release I.4 Pascal manual edited by K. A. Shillington.

Dansk Data Elektronik ApS reserves the right to change the specifications in this manual without warning. Dansk Data Elektronik ApS is not responsible for the effects of typographical errors or other inaccuracies in this manual, and cannot be held liable for the effects of the implementation and use of the structures described herein.

2. Getting started

The Pascal system package delivered by Dansk Data Elektronik ApS consists of the following:

1) A copy of the latest edition of this manual.

2) A copy of the Pascal standard reference guide (PASCAL: User Manual and Report, by Kathleen Jensen and Niklaus Wirth, Second Edition, Springer Verlag New York Inc.).

3) A floppy disc labelled "Pascal demonstration disc", which contains a number of Pascal demonstration programs (source text and P-code). For a description of these programs, please refer to the source text. Although these programs are carefully tested, Dansk Data Elektronik ApS does not guarantee their correct performance.

4) Two minifloppy discs labelled "Pascal system disc" containing the latest version of the DDE Pascal system. These floppy discs contain the following modules:
   - the standard interpreter (INTER)
   - the Pascal compiler interpreter (PASCAL)
   - the P-code for the Pascal compiler (PASCAL)
   - the relocatable code for the standard interpreter modules INTER, ARITH, VARS, PROC, STAPR, PCOMX, PFILS, FSYS, SET, FP, INITP, PNAVN, and PTALK; and the relocatable code for the non-standard interpreter modules PCOMP, FSYSX and FPX.
   - the relocatable code for the standard MIKADOS modules SOPEN, CREAT, and POSN required to link the interpreter.
   - the source code of the main interpreter module (INTER).
   - the source code of two Pascal routines:
     1) READREAL, used to convert real numbers from ASCII to binary on input from a text file
     2) WRITEREAL, used to convert real numbers from binary to

ASCII on output from a text file
A description on how to use these routines may be found in sections 7.12 and 7.13.
- the standard MIKADOS Editor (EDIT)
- two work files used by the editor (1EDITFIL and 2EDITFIL)

In order to use the MIKADOS Pascal system you must be familiar with the standard MIKADOS Editor (described in the manual "MIKADOS Editor User´s Guide", available from DDE), and with the Pascal language.

If you are familiar with Pascal and with the MIKADOS Editor we suggest that you start getting familiar with MIKADOS Pascal by reading sections 3, 4.0, 4.1, and 7.1 in this manual. This will enable you to write, edit, compile and run simple Pascal programs on your DDE computer.

If you are using Pascal E (extended precision Pascal), you should also read sections 9.1 and 9.4 before attempting to use the Pascal system.

Example: to compile and run the demonstration program "RECURSIV" (very similar to the program in example 11.9 in J&W), which may be found on the "Pascal demonstration disc", proceed as follows:

1) insert the "Pascal demonstration disc" in the P2 disc drive
2) insert the "Pascal system disc" in the P1 disc drive
3) enter >PASCAL,RECURSIV to compile the program "RECURSIV"
4) after compilation is finished, run the program "RECURSIV" by entering >INTER,RECURSIV
   Note: >RECURSIV will not run the program

3. System overview

The Pascal system consists of three programs: the standard MIKADOS editor, the Pascal compiler, and the P-code interpreter.

The standard MIKADOS editor is used to enter and edit Pascal source programs in source files stored on magnetic media (floppy discs or moving head discs). The MIKADOS editor is described in a separate publication entitled "MIKADOS Editor User´s Guide", which is available from DDE.

The Pascal compiler accepts source programs written in the standard Pascal high level computer language (with minor restrictions and extensions) and translates them into code for a hypothetical computer known as the "P-machine". The code produced by the compiler is often referred to as "P-code".

The translated Pascal program may be executed using a P-code interpreter, which simulates the hypothetical P-machine on an ID-7000 or SPC/1 computer running under the multiprogrammed MIKADOS operating system.

The compilation of a Pascal source program produces a P-code file. If the compiled Pascal program does not reference any external procedures then the P-code file may be executed immediately after the compilation without linking using the standard interpreter (INTER) delivered by DDE. If references to external procedures (usually assembler subroutines) occur in the Pascal program, a relocatable file is also produced. In this case a new version of the interpreter must be linked which includes the external procedures referenced by the Pascal program (see chapter 8).

The Pascal compiler is written in Pascal. Thus, a Pascal compilation is an interpretation of the P-code file resulting from the compilation of the Pascal compiler by itself.

The MIKADOS P-machine is a modified version of the P2 compiler interpreter originally developed by a team at the Eidgenoessische Technische Hochschule in Zuerich, Switzerland. The architecture of the P-machine is without interest to most users of the MIKADOS Pascal system as its instruction set is quite complicated and unsuitable for programming and debugging purposes. However, interested users may obtain a description of the P-machine instruction set and general architecture by contacting DDE.

4. Using the system

The execution of a Pascal program may be started in 3 ways:

1) Using the normal MIKADOS program start-up command to start the interpreter (INTER), which then loads a P-code file and executes it.
   Example:

   ```
   >INTER,QUEENS,OPTION1,OPTION2
   ```

   which executes a Pascal program named ´QUEENS´ and passes the parameter string ´OPTION1,OPTION2´ to the Pascal program

2) Using the MIKADOS program start-up command to start a special interpreter, which then loads the appropriate P-code file and executes it. This is possible only if the external reference option was specified when the Pascal program was compiled.
   Example:

   ```
   >ACCOUNTS,A,2,C,D EFG
   ```

   which executes a Pascal program named ´ACCOUNTS´ and passes the parameter string ´A,2,C,D´ to the Pascal program (´ EFG´ is ignored because of the preceding blank, see section 4.3)

3) Using the CHAIN procedure (see section 5.4.9)

Section 4.3 describes how to access the parameter string from a Pascal program.

4.1 Running the compiler

To start the Pascal compiler the user inserts the "Pascal system disc" in the P1 drive, presses ´ESC´ (´ENTER´ on some terminals), and enters:

```
>PASCAL,programname,listoption,externaloption,Pn
```

or

```
>INTER,PASCAL,programname,listoption,externaloption,Pn
```

Note: the last form can only be used if the standard interpreter includes the special compiler module, see section 4.4. The standard interpreter delivered by DDE does not include the special compiler module.

programname is the file name of the source module to be compiled. The file name must be a legal MIKADOS file name. The file name may be followed by a ´:´ and the identification of the disc on which the file resides, e.g. PROGR:P3. If a disc identification is not given all discs are searched in the order P1, P2 ... until the file is found. The source file must be a type K file.

listoption may be used to control the list output produced by the compiler.
The listoptions recognized by the compiler are T, L, and Q.
The listoption may be changed anytime during a compilation using the L and Q compile time options (see section 4.2).

L,T If L or T is specified as the listoption the compiler will produce a source listing of the

compiled program on the list device specified by the last >.LI command entered before the compiler was started. The list output includes the number of each source line compiled as well as the block level (number of RECORDs, BEGINs and statement CASEs minus number of ENDs processed) before the line is compiled. The output line never exceeds 80 characters. If a source line does not fit within the 80 characters, it is divided. Error messages will appear on the list device just after the line in which the error was detected.

T If T is specified as the listoption the compiler will output in addition to the above mentioned after each procedure information about the size of the compiled procedure (number of bytes of p-code generated), accumulated segment size (bytes of p-code), and size of temporary stack area that will be allocated for local parameters etc. by the procedure (in bytes).

Q All output except for the compiler identification messages may be suppressed by specifying the Q listoption or the Q compile time option.

Nil If an illegal listoption is specified or if no listoption is specified, the compiler will use the console device to keep the operator informed about the progress of the compilation by displaying the source line number and the name of the procedure currently being compiled. Error messages will appear on the console device (not on the >.FE device) together with a printout of the line in which the error was detected.
Only the first error detected on a source line is reported with an error message.

externaloption indicates if external procedures are referenced in the source program to be compiled.
If this option is not specified, external procedures are illegal in the source program.
The externaloptions recognized by the compiler are
- E - compile program referencing external procedures; the interpreter modules required to execute the resulting program are those included in the standard interpreter
- C - same as E, except special compiler interpreter module is required to execute the resulting P-code
- F - same as E, except floating point interpreter module is required to execute the resulting P-code
- D - same as E, except interpreter module for the SEEK standard procedure is required to execute the resulting P-code

One or more of the above options may be specified. The C, F, and D options are used to override standard interpreter configurations. Note that the resulting interpreter will always contain the modules included in the standard interpreter, i.e. the C, F, and D options are only required if the corresponding module is not included in the standard interpreter (see section 4.4).
Illegal externaloptions are ignored.

Pn is the disc identification of the disc where the P-code file and eventually the relocatable file generated by the compiler should be placed.
If this option is not specified the files are placed on disc P2.

If an option is not specified while one or more of the following options are specified, the absence of the option must be indicated by one or more commas as shown in the following example:

```
>PASCAL,MYPROG:P4,,E,P1
```

i.e. compile the source module MYPROG that resides on disc P4 and place the relocatable module on P1 instead of the default (P2). External references are allowed.

The P-code and the relocatable module (if any) produced by the compiler are placed in type P and R files, respectively, with the same name as the source file. If one or both files exist when the compiler is started, the existing contents are overwritten. If one or both files do not exist, they are created by the compiler (basic file size is 10 sectors for the P-code file and 5 sectors for the relocatable file).

No attempt should be made to execute programs containing syntactical errors.

Compilation may be aborted at any time by entering a >.BR (break) command on the console that was used to start the compilation.

4.2 Compile time options

Compile time options are set according to a convention described on pages 100-102 of J&W, where compile time options are set by means of special "dollar sign" comments inside the Pascal program text, e.g.

```
(*$option,option,....*)
```

The syntax used in the MIKADOS Pascal compiler´s control comments is essentially as described in J&W. However, the actual options and the letters associated with those options bear only occasional resemblance to the options listed in J&W. Except for the I and P options each option must be specified as a capital letter followed by "+" or "-". Illegal syntax in a control comment is not reported by the compiler but the results are unpredictable.

The following options are currently available:

C causes the compiler to generate i/o check instructions after each statement which performs any i/o. The instruction checks to see if the i/o operation was accomplished successfully. In the case of an unsuccessful i/o operation (MIKADOS file system result code >0) the program will be terminated with a user i/o error message

C+ i/o check instructions are generated (default)
C- i/o check instructions are not generated

D causes the compiler to generate line numbers in the P-code. If a run time error occurs the interpreter will print the number of the source line corresponding to the

code that was executed when the error occurred. Each line number generated occupies 3 bytes of P-code

D+ line numbers are generated (default)
D- line numbers are not generated

G determines whether Pascal GOTO statements are allowed within the program. This option may be used to restrict novice programmers from using the GOTO statement in situations where structured constructs like FOR, WHILE, REPEAT, CASE and EXIT statements would be more appropriate

G+ allows the use of the GOTO statement
G- causes the compiler to generate a syntax error upon encountering a GOTO statement (default)

I includes a source file into the compilation. The characters between ´I´ and the terminating ´*)´ are taken as the file name of the source file to be included. The comment must be closed at the end of the file name, therefore no other options can follow the file name.

Example: (*$D-,ISTRUCTURES*)

The compiler cannot keep track of nested inclusions, i.e. an include file may not have an include file control comment. This will result in a fatal syntax error.

The compiler will also relax the requirements of the order in which declarations must be made for included files which contain CONST, TYPE, VAR, PROCEDURE and FUNCTION declarations even though the original program

has previously completed its declarations. To do so the include compiler control comment must appear between the original program's last VAR declaration and the first of the original program's PROCEDURE or FUNCTION declarations.

L controls whether the compiler will generate a program listing of the following source text. This control comment is analogous to the L listoption discussed in section 4.1. The default value of this option is set by the listoption in the MIKADOS RUN command

L+ start output of source listing on the >.LI device
L- stop output of source listing

P causes the compiler to skip to a new page on the line printer if the compiler is generating a source listing on the line printer at the time when the P control comment is encountered

Q is the "quiet compiler" option which can be used to suppress the output to the console device of procedure names and line numbers detailing the progress of the compilation. This control comment is analogous to the Q listoption discussed in section 4.1. The default value of this option is set by the listoption in the MIKADOS RUN command

Q+ causes the compiler to suppress output to the console device
Q- causes the compiler to output procedure names and line numbers as well as error messages to the console device

R This option controls whether the compiler will output additional code to perform checking on array and string subscripts, and assignments to variables of subrange types

R+ turns range checking on (default)
R- turns range checking off

Programs compiled with the R- option set will run slightly faster and require less code; however, if an invalid index occurs or an invalid assignment is made, the program will not be terminated with a run time error. Until a program has been completely tested and is known to be correct, it is usually best to compile with the R+ option set. Note that certain string indexing errors (index<0 or >255) are detected even if range checking is disabled

4.3 Passing parameters to a Pascal program

When a Pascal program is started using a MIKADOS program start-up command or the CHAIN procedure, the user may pass a character string from the command or procedure call to the running program (´OPTION1,OPTION2´ in the first example in section 4.0).

This section describes how the program accesses the passed parameter string.

The user must make the following declarations:

```
TYPE  PARMARRAY = PACKED ARRAY(. 1..40 .) OF CHAR;

VAR  PARM: ^PARMARRAY;
```

The VAR declaration must be the first VAR declaration made in the program. After startup, PARM^(1..39) will contain the parameter string. The parameter string consists of the characters following the comma after the name of the Pascal program in the start command to and including the first blank encountered. A maximum of 39 characters are transferred. The first character after the parameter string in PARM^ will always be a blank. No blanks can appear within the parameter string.

4.4 Generating a new standard interpreter

The standard interpreter delivered by DDE on the Pascal system disc (see section 2) includes the SEEK procedure module and the floating point module, which may not be required by some installations. To gain more space for the execution of a compiled Pascal program an installation may choose to change the standard interpreter configuration.

To accomplish this a recompilation of the main interpreter module (INTER) on the system disc is required. First INTER is changed, using the MIKADOS Editor, according to the following rules:

1) to eliminate the floating point module (savings of approximately 1060 decimal bytes) change the line ´ EXT FP ´ into ´ EXT FPX ´. Note that interpreters without the floating point module cannot be used to compile Pascal programs containing real constants

2) to include the special compiler module (requires approximately 700 decimal bytes) change the line ´ EXT PCOMX ´ into ´ EXT PCOMP ´. Note that interpreters without the special compiler module cannot be used to compile Pascal programs

3) to eliminate the module containing the standard SEEK procedure (savings of approximately 800 decimal bytes) change the line ´ EXT FSYS ´ into ´ EXT FSYSX ´.

The user may also change the top-of-stack address (i.e. the highest memory location referenced by the interpreter) by substituting a new value in the ´MAXADR::EQU ´ statement. Changing this value will also change the size of available memory for compiling and executing Pascal programs. The value

of MAXADR must be odd or zero. Changing MAXADR to OEFFF or less will ensure that execution of a Pascal program does not destroy the Debugger program. If MAXADR is set to 0, the interpreter will use all the memory installed in the region in which the program is executed.

After performing the necessary changes, compile and link the new INTER module. Insert a MIKADOS program disc in P1 and the Pascal system disc in P2. Enter

```
>ASM,INTER
>LINK,INTER,R1,,P2
```

The new standard interpreter will replace the old one.

5. System intrinsics

This chapter describes a number of system procedures and functions that are built into the interpreter. These procedures may be used without declarations.

Most of the intrinsics assume that users are fluent in the use of Pascal and are experienced in the use of the system. Since some of these intrinsics do no checking for range validity, they may easily cause the system to crash.

5.1 String handling

In order to maintain the integrity of the length of a string, only string functions or full string assignments should be used to alter strings. Moves and/or single character assignments do not affect the length of a string which means it probably becomes wrong. The individual elements of a string are of type CHAR and may be indexed 1..LENGTH(STRING). Accessing the string outside this range will have unpredictable results if range-checking is off or cause a run-time error if range checking is on.

Strings are discussed in detail in section 7.10.

5.1.1 Length of string

```
FUNCTION LENGTH( ACTUALSTRING: STRING ): INTEGER;
```

Returns the integer value of the current length of the ACTUALSTRING.

```
Example:      GEESTRING := ´1234567´;
              WRITELN(LENGTH(GEESTRING),´   ´,LENGTH(´´));

will print    7  0
```

5.1.2 Locate pattern in string (POS)

```
FUNCTION POS( PATTERN, SOURCE: STRING ): INTEGER;
```

This function returns the position of the first occurrence in SOURCE of the pattern in PATTERN. The integer value of the position of the first character in the matched pattern will be returned; or if the pattern was not found, zero will be returned.

```
Example:      SONG := 'ROLL ME OVER IN THE CLOVER';
              PATTERN := 'VER';
              WRITELN( POS(PATTERN,SONG) );

will print    10
```

5.1.3 Concatenate strings

```
FUNCTION CONCAT( SOURCE1, SOURCE2, ... : STRING ): STRING;
```

This function returns a string which is the concatenation of all the strings passed to it. There may be any number of SOURCE strings separated by commas.

```
Example:      TEXT1 := 'WE HOLD';
              TEXT2 := 'THESE TRUTHS TO BE ';
              TEXT2 := CONCAT(TEXT1,' ',TEXT2,'SELF EVIDENT');
              WRITELN( TEXT2 );

will print    WE HOLD THESE TRUTHS TO BE SELF EVIDENT
```

5.1.4 Copy string

```
FUNCTION COPY( SOURCE: STRING; INDEX, SIZE: INTEGER ): STRING;
```

Returns a string containing SIZE characters copied from SOURCE starting at the INDEXth position in SOURCE.

Example:
```
DATE := ´TODAY IS WEDNESDAY´;
WEEKDAY := COPY( DATE, POS(´IS ´,DATE)+3, 9);
WRITELN( WEEKDAY );
```

will print WEDNESDAY

5.2 Input/output

This section describes the intrinsics used to access the MIKADOS file system from a Pascal program. A thorough discussion of the MIKADOS file system may be found in the "MIKADOS User´s Guide" manual, which is available from DDE.

In several i/o intrinsics a FILENAME is used to identify a particular file. The FILENAME must be of type STRING. The syntax of a FILENAME is

<MIKADOS filename> : <disc identification> : <size of primary file extent> : <file type>

where

<MIKADOS filename> is 1 to 8 printable characters; it is recommended that only alphanumeric characters are used in file names

<disc identification> is P1, P2, ... If a disc identification is not specified, all discs are searched until the file is located

<size of primary file extent> specified in number of 256-byte sectors

<file type> MIKADOS file type. If no file type is specified, K (source file) is assumed

After each file system operation the Pascal system checks if the error code returned by the MIKADOS system was zero. If the error code was greater than zero, a run-time error is reported and execution terminates. If i/o checking was disabled when

the program was compiled (see section 4.2), the user program must call the intrinsic function IORESULT after each file system operation to determine if the operation completed successfully. After a MIKADOS file system error has been reported, further operations on the corresponding file may have unpredictable results. MIKADOS and Pascal error codes are explained in appendix A.

There are three predeclared files: INPUT (console terminal input), OUTPUT (console terminal output), and LIST (list device, controlled by the >.LI command). The predeclared files are TEXT (FILE OF CHAR) files. No attempt should be made to RESET, REWRITE or CLOSE these files.

In the following sections PHYLE is a type designator, either

```
        TYPE    PHYLE = FILE OF <type>;
or      TYPE    PHYLE = TEXT;
```

5.2.1 Open/create file

PROCEDURE RESET(FILEID: PHYLE);

This procedure resets the position of a currently open file to its beginning for the purpose of reading. EOF(FILEID) is set to false, and EOLN(FILEID) is set to true. FILEID^ points to a blank in TEXT files. A READLN(FILEID) or GET(FILEID) must be issued before the first record in the file can be accessed.

PROCEDURE RESET(FILEID: PHYLE; FILENAME: STRING);

This procedure opens a previously existing file identified by FILENAME. Disc identification, file size and file type may be omitted in the FILENAME in which case the default values are used. The file size is always ignored. The opened file is positioned to its beginning for the purpose of reading. Otherwise RESET(FILEID, FILENAME) operates exactly as RESET(FILEID).

PROCEDURE REWRITE(FILEID: PHYLE; FILENAME: STRING);

This procedure opens a previously existing file identified by FILENAME and positions it to its beginning for the purpose of writing (and reading). A disc identification must always be specified in FILENAME. If the file identified by FILENAME cannot be located, it is created according to the parameters in FILENAME. In this case the size of the primary extent must be specified.

EOF(FILEID) is set to false. EOLN(FILEID) is undefined.

As specified in J&W, RESET and REWRITE should not be applied to the predeclared files INPUT, OUTPUT, and LIST.

5.2.2 Close file

```
PROCEDURE CLOSE( FILEID: PHYLE );
```

Marks the file specified by FILEID closed. The implicit variable FILEID^ is made undefined. CLOSEing a closed file causes no action.

All files declared within a procedure (program) are closed automatically by the system when the procedure (program) is exited.

CLOSE should not be applied to the predeclared files INPUT, OUTPUT, and LIST.

5.2.3 Check for end-of-file/end-of-line

```
FUNCTION EOF ( FILEID: PHYLE ): BOOLEAN;
FUNCTION EOLN( FILEID: PHYLE ): BOOLEAN;
```

These functions are similar to those defined on page 160 of J&W. However, the user should note the supplementary comments in sections 7.15 and 7.16.

5.2.4 Determine result of i/o operation

```
FUNCTION IORESULT: INTEGER;
```

After any i/o operation, IORESULT returns the resulting MIKADOS file system error code as an integer value. The meaning of the MIKADOS file system error codes is explained in appendix A.

Use of the IORESULT function is required only in programs where i/o-checking has been disabled using the C- control comment (see section 4.2), or to detect certain non-serious errors (error code < 0, see appendix A).

5.2.5 Read/write record from direct access file

```
PROCEDURE GET( FILEID: PHYLE );
PROCEDURE PUT( FILEID: PHYLE );
```

These procedures are similar to those defined on page 158 of J&W. However, the user should note the supplementary comments in section 7.14.

5.2.6 Read/write record from sequential file

```
PROCEDURE READ( FILEID: PHYLE; <argument list> );
PROCEDURE READLN( FILEID: PHYLE; <argument list> );
PROCEDURE WRITE( FILEID: PHYLE; <argument list> );
PROCEDURE WRITELN( FILEID: PHYLE; <argument list> );
```

These procedures are similar to those defined on page 161 - 163 of J&W. However, the user should note the supplementary comments in section 7.12 and 7.13.

5.2.7 Eject page (PAGE)

```
PROCEDURE PAGE( FILEID: PHYLE );
```

This procedure, as described in J&W (page 164), sends a top-of-form command (´<ES>´) to the file.

This procedure should be applied only to files of type TEXT.

5.2.8 Position to record in direct access file (SEEK)

```
PROCEDURE SEEK( FILEID: PHYLE; RECORDNUMBER: INTEGER );
```

This procedure changes the file pointers so that the next GET or PUT from/to the file will happen to the RECORDNUMBERth record of FILEID. Records in files are numbered from 1.

This procedure should not be applied to files of type TEXT.

Note that the record number is automatically increased by 1 by each GET or PUT issued to a direct access file. Updating a particular record in a direct access file consequently requires <u>two</u> seeks, i.e.

```
SEEK( FILE, 5 );
GET( FILE );
...
(* update record *)
...
SEEK( FILE, 5 ); (*change record no. from 6 to 5*)
PUT( FILE );
...
(*a  GET here would read in record no. 6 *)
```

5.2.9 Edit string on output device

PROCEDURE EDIT(<editstring>);

```
where  <editstring> ::= <stringidentifier>
or     <editstring> ::= <stringidentifier> : <editlength>
       <editlength> ::= <integer expression>
```

This procedure is used to edit (update) a textstring, e.g. a previously entered name or address.

The procedure outputs the current contents of the string identified by <stringidentifier> to the console device (OUTPUT unit). After the string has been output the user may edit it using the standard MIKADOS edit functions (cursor forward and backward, insert and delete character, rubout and erase to end of line). The user terminates the operation by pressing RETURN or ESC, which causes an immediate return from the EDIT

procedure to the Pascal program. Upon return the string identified by <stringidentifier> contains the edited string. EOF(INPUT) = TRUE if ESC was pressed.

The ´: <editlength>´ is optional and indicates the number of characters in the string to be edited. If this parameter is not specified, the editlength is taken to be the current dynamic length of the string.

If the editlength is greater than the current dynamic length of the string, the previously undefined string characters are blanked by the interpreter before the edit operation. After the edit operation the dynamic length of the string edited is set equal to the editlength.

If the editlength is specified as being greater than the maximum size of the string, a run time error occurs. If the editlength is zero, or if the editlength is not specified and the length of the string is zero, then the EDIT procedure returns immediately to the Pascal program, and no i/o operation takes place.

The value returned by IORESULT is not affected by EDIT.

Example:
```
PROGRAM EDITDEMO;
  VAR ANSWER: STRING(9);
BEGIN
REPEAT
  BEGIN
  WRITE(´Do you want to terminate ? ´);
  ANSWER = ´No       ´;
  EDIT( ANSWER );
  (* Note: program suggests answer ´No´;
     user may accept just by pressing RETURN *)
  END;
 UNTIL (ANSWER=´Yes      ´) OR (ANSWER=´Yeah, man´);
 END (*EDITDEMO*).
```

5.2.10 Clear screen on output device

PROCEDURE CLEARSCREEN;

This procedure clears the screen on the console device (OUTPUT unit).

The screen is cleared by outputting a ´<XS>´ string to the console driver.

5.3 Character array manipulation

These intrinsics are all byte oriented. Use them with care; no range checking of any sort is performed on the parameters passed to these routines.

The intrinsic SIZEOF (defined in section 5.4.1) is meant for use with these intrinsics; it is convenient not to have to figure out or remember the number of bytes in a particular data structure.

The type PACKD used in several declarations of intrinsic procedures in this section is defined as

```
TYPE PACKD = PACKED ARRAY(0..N) OF CHAR;
```

5.3.1 Scan character array

```
FUNCTION SCAN( LENGTH: INTEGER; <partial expression>;
                                        ARRAY: PACKD ): INTEGER;
```

This function returns the number of characters from the starting position to where the scan was terminated. The scan terminates on either matching the specified LENGTH or satisfying the <partial expression>. The ARRAY should be a packed array of characters and may be subscripted to denote the starting point. If the expression is satisfied on the character at which ARRAY is pointed, the value returned will be zero. If the length passed was negative, the number returned will also be negative, and the function will have scanned backward. The <partial expression> must be of the form

´<>´ or ´=´ followed by a <character expression>

Examples: Using the array COUNT := ´0000123456789ABCDEFGHI´

```
SCAN( -10, =´ ´, COUNT(.17.) )   will return -10
SCAN( 100, <>´0´, COUNT )        will return 4
SCAN( 15, =´9´, COUNT(.0.) )     will return 12
SCAN( -7, =´C´, COUNT(.10.))     will return -5
```

5.3.2 Move character array (MOVELEFT, MOVERIGHT)

```
PROCEDURE MOVELEFT ( SOURCE: PACKD;
                     VAR DESTINATION: PACKD;
                     LENGTH: INTEGER );
PROCEDURE MOVERIGHT( SOURCE: PACKD;
                     VAR DESTINATION: PACKD;
                     LENGTH: INTEGER );
```

These procedures do mass moves of byte strings of the LENGTH specified. MOVELEFT starts from the left end of the specified SOURCE and copies bytes to the left end of the DESTINATION traveling right. MOVERIGHT starts from the right end of both arrays and copies bytes traveling left. The reason for having both procedures is that if the source and destination arrays overlap the order in which characters are moved is critical. The following examples show what happens if you use the procedure which moves in the wrong direction for your purpose.

Examples:

```
ORIGINAL := ´aabcdefghijklmnop´;
TEXT := ORIGINAL;
MOVELEFT( TEXT(.10.), TEXT(.2.), 5);
    (*TEXT:  aijklmfghijklmnop*)
TEXT := ORIGINAL;
MOVERIGHT( TEXT(.2.), TEXT(.1.), 5);
    (*TEXT:  aeeeeefghijklmnop*)
TEXT := ORIGINAL;
MOVELEFT(  TEXT(.2.), TEXT(.1.), 5);
    (*TEXT:  abcdeefghijklmnop*)
```

5.3.3 Initialize character array (FILLCHAR)

```
PROCEDURE FILLCHAR( VAR DESTINATION: PACKD; LENGTH: INTEGER;
                                            CHARACTER: CHAR );
```

This procedure takes a (subscripted) packed array of characters and fills it with the number (LENGTH) of CHARACTERs specified. The procedure is equivalent to

```
DESTINATION(.0.) := CHARACTER;
MOVELEFT( DESTINATION(.0.), DESTINATION(.1.), LENGTH-1 );
```

but FILLCHAR is twice as fast.

5.4 Miscellaneous useful routines

5.4.1 Determine size of structure (SIZEOF)

FUNCTION SIZEOF(<variable or type identifier>): INTEGER;

This function returns the number of bytes that the ´item´
passed as a parameter occupies on the stack. SIZEOF is
particularly useful in connection with the FILLCHAR, MOVELEFT
and MOVERIGHT intrinsics.

5.4.2 Read system clock (TIME)

FUNCTION TIME: INTEGER;

This function returns the current value of the system clock in
units of 10 milliseconds as an unsigned 16-bit integer. The
clock is updated by the MIKADOS operating system once every
10 ms - 5 sec depending on the hardware configuration. The
interval between clock updates may vary from system to system.
Clock overflow is ignored. The absolute value of the returned
integer is without significance; however, the differences
between the integers returned by two subsequent calls of TIME
may be used as a measure of the time elapsed between the two
calls.

5.4.3 Compute power of ten (PWROFTEN)

```
FUNCTION PWROFTEN( EXPONENT: INTEGER ): REAL;
```

This function returns the value of 10 to the EXPONENT power. EXPONENT must be an integer in the range 0..37.

5.4.4 Dynamic memory control (MARK, RELEASE)

```
PROCEDURE MARK( VAR HEAPPTR: ^INTEGER );
PROCEDURE RELEASE( VAR HEAPPTR: ^INTEGER );
```

These procedures are used for returning dynamic memory allocations to the system. MARK sets HEAPPTR to the current top-of-heap. RELEASE sets the top-of-heap pointer to HEAPPTR.

Further details about these routines and their use may be found in section 7.3.

5.4.5 Move cursor (GOTOXY)

```
PROCEDURE GOTOXY( XCOORD, YCOORD: INTEGER );
```

This procedure moves the cursor on the console terminal to the coordinates specified by (XCOORD, YCOORD). The upper left corner of the terminal screen is assumed to be (1,1). The lower right corner of the terminal screen is usually (80,24), but may vary according to the number of characters per line and the number of lines on the actual display terminal.

5.4.6 Basic input/output operations (IN80, OUT80)

```
FUNCTION IN80( IOADDRESS, MASK: INTEGER ): INTEGER;
PROCEDURE OUT80( IOADDRESS, VALUE: INTEGER );
```

These routines perform basic INTEL 8080 input and output operations on specified i/o addresses (ports).

A call of the IN80 function is equivalent to the assembler sequence

```
        IN      IOADDRESS
        ANI     MASK
        STA     IN80
```

i.e. 8 bits are read from the i/o address specified, the 8 bits are AND´ed with the MASK and the result is returned as the function value.

A call of the OUT80 procedure is equivalent to the assembler sequence

```
        LDA     VALUE
        OUT     IOADDRESS
```

i.e. the least significant 8 bits of VALUE are output to the i/o address specified.

IOADDRESS, MASK and VALUE must be integer expressions. The most significant 8 bits of the expression values are ignored.

These operations are not protected in any way. They should be used only by users who are familiar with the ID-7000 and SPC/1 basic input/output system. Details about i/o addresses and the meaning of input and output bit patterns may be obtained from the appropriate DDE hardware manuals. Under no circumstances should a user issue OUT80 procedure calls to a device that is being controlled by a MIKADOS driver.

5.4.7 Set i/o result

PROCEDURE SETIORESULT(NEWIORESULT: INTEGER);

This procedure sets the value of the internal i/o result variable to the value specified by the integer expression NEWIORESULT. The value may subsequently be read by a call to the IORESULT function (see section 5.2.4).

This procedure provides a convenient way of communicating i/o result codes between user-written i/o procedures and application programs.

5.4.8 Delay program

PROCEDURE DELAY(DELAYTIME: INTEGER);

This procedure delays the process executing the current Pascal program for a time period specified through the DELAYTIME integer expression. The 16-bit unsigned value resulting from the computation of the DELAYTIME expression specifies the number of 10-ms periods that must elaps before execution of the program is resumed.

Note that in many MIKADOS systems the timer resolution is coarser than 10 ms.

5.4.9 Start new program (CHAIN)

```
PROCEDURE CHAIN( PROGRAMNAME, PARMSTRING: STRING );
```

This procedure uses the MIKADOS ´Start a Process´ call to initiate a new process with the execution of the program whose name is contained in PROGRAMNAME. The parameter string contained in PARMSTRING is passed to the new process.

The dynamic length of PROGRAMNAME must be exactly 10 characters. Characters 1 - 8 must contain the program name. Character 9 must contain a ´*´ if the new process should execute in the same region and bank as the initiating process, otherwise character 9 should contain

CHR(256*bank number + binary region number).

Character 10 must contain the second character in the disc identification of the disc where the program is located, i.e. ´2´ if the program resides on disc P2.

Only the first 39 characters of the parameter string in PARMSTRING are transferred to the new program. The parameter string is accessed by the new program using either the VINIT subroutine (see the ´MIKADOS Utility Programs and Subroutines´ manual), or the method described in section 4.3.

The new process will be assigned the standard priority 6.

Example: starting a Pascal compilation from a Pascal program (analogous to the MIKADOS >PASCAL,MYPROG:P4,,E,P1 command, see section 4.1):

```
CHAIN( ´PASCAL  *1´, ´MYPROG:P4,,E,P1 ´ );
```

6. Segment procedures and functions

Declarations of segment procedures and functions are identical to declarations of Pascal procedures and functions except they are preceded by the reserved word ´SEGMENT´, for example:

```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
  (* Pascal code *)
END;
```

Program behavior differs, however, in that code and data for a segment procedure (function) are in memory only while there is an active invocation of that procedure.

The user may put large pieces of one-time code, e.g. initialization code, into a segment procedure. After performing the initialization, the memory area of the now useless code is released thus increasing the available memory space.

The disc which holds the codefile for the program must be online whenever a new segment procedure is to be called. A maximum of 15 segment procedures are ordinarily available to the user. Segment procedures must be the first procedure declarations containing code-generating statements in a program.

The following program is an example of the use of SEGMENT PROCEDURES:

```
PROGRAM SEGMENTDEMO;

(* GLOBAL DECLARATIONS GO HERE *)

PROCEDURE PRINT( T: STRING ); FORWARD; (*DONT GENERATE CODE YET*)

SEGMENT PROCEDURE ONE;
BEGIN
  PRINT(´SEGMENT NUMBER ONE´)
END;

SEGMENT PROCEDURE TWO;
  SEGMENT PROCEDURE THREE;
  BEGIN
    ONE;
    PRINT(´SEGMENT NUMBER THREE´)
  END;
BEGIN (*SEGMENT NUMBER TWO*)
  THREE;
  PRINT(´SEGMENT NUMBER TWO´)
END;

PROCEDURE PRINT;
BEGIN
  WRITELN(OUTPUT,T)
END;

BEGIN
  TWO;
  WRITELN(´I´´M DONE´)
END.
```

The above program will give the following output:

```
SEGMENT NUMBER ONE
SEGMENT NUMBER THREE
SEGMENT NUMBER TWO
I´M DONE
```

7. Differences between MIKADOS Pascal and standard Pascal

This chapter contains a description of the various differences between MIKADOS Pascal and standard Pascal. The standard Pascal referred to is defined in ´PASCAL user manual and report´ (2nd edition) by Kathleen Jensen and Niklaus Wirth, Springer Verlag 1978. The standard is referred to as J&W in the following discussion.

A number of differences and extensions are treated in other chapters of this manual. These are

- segment procedures and functions (chapter 6)
- external procedures (chapter 8)

It is recommended that the reader first concentrate upon the sections of this chapter which describe the differences associated with the character set.

7.1 Character set and special symbols

The character set used in MIKADOS Pascal differs from that defined in J&W on page 137 in the following ways:

1) <letter> has been extended to include the national Danish characters Æ, Ø, Å, æ, ø, and å

2) upper and lower case characters are considered equivalent. For example, the identifiers XYZ and xYz are equivalent, and the keyword BEGIN may also be written Begin or begin.

3) the left and right brackets [and] used in subscripts and sets have been replaced by the symbols ´(.´ and ´.)´ (the character values associated with the brackets are

used for national characters). However, in order to improve the readability of MIKADOS Pascal programs it is allowed to use ´(´ and ´)´ instead of ´(.´ and ´.)´ around array indices. The <element list> in a <set> must be surrounded by ´(.´ and ´.)´.

4) the left and right brackets { and } cannot be used to delimit comments (the character values associated with these brackets are used for national characters); comments may be written only between the symbols (* and *)

5) SEGMENT, FORWARD, and EXTERNAL are reserved keywords

7.2 CASE statements

J&W on page 31 state that if there is no label equal to the value of the case statement selector, then the result of the case statement is undefined. MIKADOS Pascal in this case will execute the statement immediately following the case statement.

Contrary to the syntax diagrams for <field list> on page 116 of J&W, the MIKADOS Pascal compiler will not permit a semicolon before the ´END´ of a case variant field declaration within a RECORD declaration. See appendix C for the revised syntax diagram for <field list>.

7.3 Dynamic memory allocation

The standard procedure DISPOSE defined on page 158 of J&W is not implemented in MIKADOS Pascal. However, the function of DISPOSE can be approximated by a combined use of the MIKADOS Pascal intrinsics MARK and RELEASE. The process of recovering memory space as described below is only an approximation to the function of DISPOSE in that one cannot explicitly ask that the storage occupied by one particular variable be released by the system for other uses.

MIKADOS Pascal allocates storage for variables created by use of the standard procedure NEW in a stack-like structure called the ´heap´. The following program is a simple demonstration of how MARK and RELEASE can be used to cause changes in the size of the heap:

```
PROGRAM SMALLHEAP
TYPE PERSON = RECORD
                NAME: PACKED ARRAY(.0..30.) OF CHAR;
                ID: INTEGER
              END;
VAR P: ^PERSON;  (* ´^´ means ´pointer to´ *)
    HEAP: ^INTEGER;

BEGIN
  MARK( HEAP );
  NEW( P );          (* ALLOCATE RECORD *)
  P^.NAME := ´FINKELSTEIN, SAM´;
  P^.ID   := 999;
  RELEASE( HEAP ) (* RELEASE SPACE OCCUPIED BY RECORD*)
END.
```

The above program first calls MARK to place the address of the current top-of-heap into the variable HEAP. The parameter

supplied to MARK must be a pointer variable, but need not be declared to be a pointer to an INTEGER as is traditional.

Next the program calls the standard procedure NEW and this results in a new variable P^ which is located in the heap as shown in the diagram below:

```
NEW TOP OF HEAP  -->    :---------------:
                        :               :
                        :       P^      :
                        :               :
                        :---------------:  <-- OLD TOP OF HEAP
                        :contents of    :
                        :heap at start  :
                        :of program     :
                        :               :
```

Once the program no longer needs the variable P^ and wishes to release this memory space to the system for other uses, it calls RELEASE which resets the top-of-heap to the address contained in the variable HEAP.

If the above program had done a series of calls to the standard procedure NEW between the calls to MARK and RELEASE, then the effect would have been that the storage occupied by several variables would have been released at once. Also note that due to the stack nature of the heap it it not possible to release the memory space used by a single item in the middle of the heap.

It should be noted that careless use of the intrinsics MARK and RELEASE can lead to pointers which point to areas of memory which are no longer a part of the defined heap space.

7.4 GOTO and EXIT statements

MIKADOS Pascal has a more limited form of the GOTO statement than is defined as the standard in J&W. MIKADOS Pascal´s GOTO statement may not transfer control to a label which is not within the same block as the GOTO statement itself. The example presented on page 32 of J&W is not legal in MIKADOS Pascal.

EXIT is a MIKADOS Pascal extension which accepts as its single parameter the identifier of a procedure or function to be exited.

If the procedure identifier passed to EXIT is a recursive procedure then the most recent invocation of that procedure will be exited. Also, local files are CLOSEd implicitly when a procedure is EXITed just as if it had terminated normally.

The creation of the EXIT statement was inspired by the occasional need for a straightforward means to abort a complicated and possibly deeply nested series of procedure calls upon encountering an error.

7.5 Packed variables

7.5.1 Packed arrays

The MIKADOS Pascal compiler will perform packing of arrays and records if the ARRAY or RECORD declaration is preceded by the word PACKED. For example, consider the following declarations:

```
A: ARRAY(. 0..9 .) OF CHAR;
B: PACKED ARRAY(. 0..9 .) OF CHAR;
```

The array A will occupy ten 16-bit words of memory, with each element of the array occupying 1 word. The PACKED ARRAY B on the other hand will occupy a total of only 5 words since each 16 bit word contains two 8-bit characters. In this manner each element of the PACKED ARRAY B is 8 bits long.

Packed arrays need not be restricted to arrays of type CHAR, for example:

```
C: PACKED ARRAY(. 0..1 .) OF 0..3;
D: PACKED ARRAY(. 1..9 .) OF SET OF 0..15;
E: PACKED ARRAY(. 0..239,0..319 .) OF BOOLEAN;
```

Each element of the PACKED ARRAY C is only 2 bits long, since only 2 bits are needed to represent the values in the range 0..3. Therefore C occupies only one 16 bit word of memory, and 12 of the bits in that word are unused. The PACKED ARRAY D is a 9 word array, since each element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of E in the above example, occupies only one bit.

The following 2 declarations are not equivalent due to the recursive nature of the compiler:

```
F: PACKED ARRAY(. 0..9 .) OF ARRAY(. 0..3 .) OF CHAR;
G: PACKED ARRAY(. 0..9,0..3 .) OF CHAR;
```

The second occurrence of the reserved word ARRAY in the declaration of F causes the packing option in the compiler to be turned off. The net result is that F becomes an unpacked array of 40 words. On the other hand, the PACKED ARRAY G is an array occupying 20 total words. If F had been declared as

```
      F: PACKED ARRAY(. 0..9 .) OF PACKED ARRAY(. 0..3 .) OF CHAR;
or as F: ARRAY(. 0..9 .) OF PACKED ARRAY(. 0..3 .) OF CHAR;
```

then F and G would have had identical configurations.

In short, the reserved word PACKED only has true significance before the last appearance of the reserved word ARRAY in a declaration of a PACKED ARRAY. When in doubt a good rule of thumb when declaring a multidimensional PACKED ARRAY is to place the reserved word PACKED before every appearance of the reserved word ARRAY to insure that the resultant array will in fact be packed.

The resultant array will only be packed if the final type of the array is scalar, boolean, CHAR, subrange, or a set which can be represented in 8 bits or less. The following declaration will not result in any packing because the final type of the array cannot be represented in a field of 8 bits:

```
H: PACKED ARRAY(. 0..3 .) OF 0..1000;
```

H will be an array which occupies 4 16-bit words.

Packing never occurs across word boundaries. This means that if the type of the element to be packed requires a number of bits which does not divide evenly into 16, then there will be some unused bits at the high order end of each of the words which comprise the array.

Note that a string constant may be assigned to a PACKED ARRAY OF CHAR but not to an unpacked ARRAY OF CHAR. Likewise, comparisons between an ARRAY OF CHAR and a string constant are illegal. Because of their different sizes, packed arrays cannot be compared to ordinary unpacked arrays.

A PACKED ARRAY OF CHAR may be output with a single write statement:

```
PROGRAM VERYSLICK;
VAR T: PACKED ARRAY(. 0..10 .) OF CHAR;
BEGIN
  T := ´HELLO THERE´;
  WRITELN( T )
END.
```

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently by using the SIZEOF and FILLCHAR intrinsics defined in sections 5.4.1 and 5.3.3, respectively.

7.5.2 Packed records

The following RECORD declaration declares a RECORD with 4 fields. The entire record occupies one 16 bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD
         I, J, K: 0..31;
         B: BOOLEAN
       END;
```

The variables I, J, K each take up 5 bits in the word. The boolean variable B is allocated in the 16th bit of the same word.

In much the same manner that packed arrays can be multidimensional, packed records may contain fields which themselves are packed records or packed arrays. Again, slight differences in the way in which declarations are made will affect the degree of packing achieved. For example, note that the following two declarations are not equivalent:

```
VAR A: PACKED RECORD
            C: INTEGER;
            F: PACKED RECORD
                   R: CHAR;
                   K: BOOLEAN
               END;
            H: PACKED ARRAY(. 0..3 .) OF CHAR
          END;

VAR B: PACKED RECORD
            C: INTEGER;
            F: RECORD
                   R: CHAR;
                   K: BOOLEAN
               END;
            H: PACKED ARRAY(. 0..3 .) OF CHAR
          END;
```

As with the reserved word ARRAY, the reserved word PACKED must appear with every occurrence of the reserved word RECORD in order for the packed record to retain its packed qualities throughout all fields of the record. In the above example, only the record A is as completely packed as possible. In B, the F field is not packed and therefore occupies two 16 bit words. In contrast A.F has all of its fields packed into one word. However, it is important to note that a packed or unpacked array or record which is a field of a packed record will always start at the beginning of the next word boundary. This means that in the case of A in the above example, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

A case variant may be used as the last field of a packed record, and the amount of space allocated to it will be the size of the largest variant among the various cases. The

actual nature of the packing is beyond the scope of this manual.

```
VAR K: PACKED RECORD
         B: BOOLEAN;
         CASE F: BOOLEAN OF
              TRUE:  (Z: INTEGER);
              FALSE: (M: PACKED ARRAY(. 0..3 .) OF CHAR)
         END
       END;
```

In the above example the B and F fields are stored in two bits of the first 16 bit word of the record. The remaining 14 bits are not used. The size of the case variant field is always the size of the largest variant, so in the above example, the case variant field will occupy two words. Thus the entire packed record will occupy 3 words.

7.5.3 Using packed variables as parameters

No element of a packed array or field of a packed record may be passed as a variable (call-by-reference) parameter to a procedure or function. Packed variables may, however, be passed as call-by-value parameters (as stated in J&W).

7.5.4 PACK and UNPACK standard procedures

MIKADOS Pascal does not support the standard procedures PACK and UNPACK as defined in J&W page 106.

7.6 Parametric procedures and functions

MIKADOS Pascal does not support the construct in which procedures and functions may be declared as formal parameters in the parameter list of a procedure or function.

See appendix C for the revised syntax diagram of <parameter-list>.

7.7 Program headings

Although the MIKADOS Pascal compiler will permit a list of file parameters to be present following the program identifier in the PROGRAM statement, these parameters are ignored by the compiler and will have no effect on the program being compiled.

7.8 The standard types

The standard types boolean, character and pointer are exactly as defined in J&W. This section describes certain implementation dependent details about integers and reals.

7.8.1 Integers

Integers are represented internally in 16 bit, two´s complement, capable of representing values in the range -32768..32767.

Integer overflow during computation is ignored. The result of an operation causing integer overflow is unpredictable.

7.8.2 Reals

Reals are represented internally in 32 bit. The detailed format is described in section 8.2. The mantissa has 23 bits corresponding to approximately 6.8 digits precision. The exponent has 8 bits, which corresponds to an exponent range of approximately 10**(-38) to 10**38 and true zero.

Real overflow during computation causes a run-time (interpreter) error. Real underflow causes the result to be set to a true zero and execution continues.

The intrinsic functions ARCTAN, COS, EXP, LN, SIN, and SQRT mentioned in J&W appendix C, page 109, are recognized by the compiler but cause a run-time error if called.

7.9 Sets

MIKADOS Pascal supports all of the constructs defined for sets on pages 50-51 of J&W. A set can be at most 255 words in size, and have at most 4080 elements.

Comparisons and operations on sets are allowed only between sets which are either of the same base type or subranges of the same underlying type. For example, in the sample program below, the base type of the set S is the subrange type 0..49, while the base type of the set R is the subrange type 0..100. However, the underlying type of both sets is the type INTEGER, which by the above definition of compatibility implies that the comparisons and operations on the sets S and R in the following program are legal:

```
PROGRAM SETCOMPARE;
VAR S: SET OF 0..49;
    R: SET OF 0..100;

BEGIN
  S := (. 0,5,10,15,20,25,30,35,40,45 .);
  R := (. 10,20,30,40,50,60,70,80,90 .);
  IF S = R THEN WRITELN(´ ...oops... ´)
  ELSE          WRITELN(´sets work´);
END.
```

However, in the following example the construct I = J is not legal since the two sets are of different underlying types.

```
PROGRAM ILLEGAL;
TYPE NUMBERS=(ZERO,ONE,TWO);
VAR I: SET OF NUMBERS;
    J: SET OF 0..2;

BEGIN
  I := (. ZERO .);
  J := (. 1,2 .);
  IF I=J THEN ;              <<<< ERROR
END.
```

7.10 Strings

MIKADOS Pascal has an additional predeclared type STRING. Variables of type string are essentially packed arrays of characters that have a dynamic length attribute, the value of which is returned by the string intrinsic LENGTH (see section 5.1.1). The default maximum length of a string variable is 80 characters. This default maximum length can be overridden in the declaration of a string variable by appending the desired length of the string variable within (. .) or ( ) after the reserved type identifier STRING. Examples of declarations of string variables appear below:

```
TITLE: STRING;    (* defaults to a maximum length of 80
                     characters *)
NAME: STRING(.20.)   (* allows the string to be a
                        maximum of 20 characters *)
```

Note that a string variable has an absolute maximum length of 255 characters. Assignments to string variables can be performed using the assingment statement, the string intrinsics, or by means of a READ statement:

```
       TITLE := '   THIS IS A TITLE    ';
or     NAME := COPY( TITLE, 1, 20 );
or     READLN( TITLE );
```

The individual characters within a string are indexed from 1 to the length of the string, for example:

```
       TITLE(.1.) := 'A';
       TITLE(. LENGTH(TITLE) .) := 'Å';
```

A variable of type string may not be indexed beyond its current dynamic length. The following sequence will result in an invalid index run time error:

```
       TITLE := '1234';
       TITLE(.5.) := '5';
```

The dynamic length of a string is accessible as the string element with index 0, but only if range checking is disabled. The following sequence will execute without error:

```
       TITLE := '1234';
       (*$R-*)  TITLE(.0.) := 5  (*$R+*);
       TITLE(.5.) := '5';
```

A variable of type string may be compared to any other variable of type string or a string constant no matter what its current dynamic length. Unlike comparisons involving variables of other types, string variables may be compared to items of a different length. The resulting comparison is

lexicographical. Lower case characters are greater than upper case characters. The following program is a demonstration of legal comparisons involving variables of type string:

```
PROGRAM COMPARESTRINGS;
VAR S: STRING;
    T: STRING(.40.);

BEGIN
  S := ´SOMETHING´;
  T := ´SOMETHING BIGGER´;
  IF S=T THEN WRITELN(´Strings don´´t work too well´);
  ELSE IF S>T THEN WRITELN(S,´ is greater than ´,T)
       ELSE IF S<T THEN WRITELN(S,´ is less than ´,T);
  IF S=´SOMETHING´ THEN WRITELN(S,´ equals ´,S);
  IF S>´SAMETHING´ THEN
       WRITELN(S,´ is greater than SAMETHING´);
  IF S=´SOMETHING ´ THEN WRITELN(´Blanks don´´t count´);
  ELSE WRITELN(´Blanks appear to make a difference´);
  S := ´XXX´;
  T := ´ABCDEF´;
  IF S>T THEN WRITELN(S,´ is greater than ´,T)
         ELSE WRITELN(S,´ is less than ´,T);
  IF ´UPPERCASE´<´lowercase´ THEN
         WRITELN(´lowercase is greater than UPPERCASE´)
  ELSE WRITELN(´Lowercase=uppercase´)
END.
```

The above program should produce the following output:

```
SOMETHING is less than SOMETHING BIGGER
SOMETHING equals SOMETHING
SOMETHING is greater than SAMETHING
Blanks appear to make a difference
XXX is greater than ABCDEF
lowercase is greater than UPPERCASE
```

When a variable of type string is a parameter to the standard procedures READ and READLN, all characters up to the end of line character in the source file will be assigned to the string variable. Note that care must be taken when reading string variables. The single statement READLN(S1,S2) is equivalent to the two statement sequence READ(S1); READLN(S2). In both cases the string variable S2 will be assigned the empty string.

7.11 Extended comparisons

MIKADOS Pascal allows = and <> comparisons of any array or record structure.

7.12 READ and READLN

The standard procedures READ and READLN are compatible with standard Pascal except for the following:

READ and READLN should be used only on sequential files (files of type TEXT or FILE OF CHAR).

When a file of type TEXT or FILE OF CHAR is opened, a record containing one blank is placed in the file window, and EOLN(FILEID) is set to true.

When a variable of type string is a parameter to the standard procedures READ and READLN, all characters up to the end of line character in the source file will be assigned to the string variable.

If variables of type REAL are to be input, the user must include in his source module the READREAL procedure, which is then called by the system to perform the input operation. READREAL is supplied by DDE on the Pascal system disc. READREAL must be included in the outermost system block (level 0) using the Include control comment described in section 4.2.

If a syntax error is encountered while READing an integer or a real, the error code returned by IORESULT is set to -3 or -2, respectively.

7.13 WRITE and WRITELN

The standard procedures WRITE and WRITELN are compatible with standard Pascal except for the following:

MIKADOS Pascal does not support the output of the words TRUE and FALSE as the result of writing out the value of a boolean variable.

If a variable is written without specifying a field length, the actual number of characters written is equal to the length of the ASCII representation of the variable. No blanks are written before or after an integer or real output with a WRITE or WRITELN.

A record is written onto a file only after it has been terminated with a WRITELN except for the OUTPUT file where output always occurs when the final parameter in the procedure call has been written in the system buffer. WRITE and WRITELN should be used only on sequential files (files of type TEXT or FILE OF CHAR).

If during a WRITE or WRITELN more than 80 characters are written in the system buffer, the 80 characters are output immediately as a separate record after which the interrupted operation continues.

If variables of type REAL are to be output the user must include in his source module the WRITEREAL procedure, which is called by the system to perform the output operation. WRITEREAL is supplied by DDE on the Pascal system disc. WRITEREAL must be included in the outermost program block (block level 0) using the Include control comment described in section 4.2.

MIKADOS Pascal´s WRITE and WRITELN does support the writing of entire PACKED ARRAYs OF CHAR in a single WRITE statement. See section 7.5 for further information about packing.

The following program demonstrates the effects of a field width specification within a WRITE statement for a variable of type string:

```
PROGRAM WRITESTRINGS;
VAR S:STRING;

BEGIN
  S := ´THE BIG BROWN FOX JUMPED...´;
  WRITELN(S);
  WRITELN(S:30);
  WRITELN(S:10)
END.
```

The above program will produce the following output:

```
THE BIG BROWN FOX JUMPED...
   THE BIG BROWN FOX JUMPED...
THE BIG BR
```

Note that when a string variable is written without specifying a field width, the actual number of characters written is equal to the dynamic length of the string. If the field width specified is longer than the dynamic length of the string, then leading blanks are written. If the field width is smaller than the dynamic length of the string then the excess characters will be truncated on the right.

7.14 PUT and GET

These procedures are similar to those defined on page 158 of J&W. The following supplementary comments should be noted:

GET(FILEID) — a GET(FILEID) is required after a RESET or REWRITE to assign the value of the first record to the buffer variable FILEID^. Switching between file extents is performed automatically.

PUT(FILEID) — the file is automatically extended as the need arises.

7.15 EOF - end-of-file

This function is similar to the EOF function defined on page 160 of J&W. The following supplementary comments should be noted:

After a RESET or a REWRITE has been issued for a file, EOF returns false. EOF returns true on a closed file. When EOF(FILEID) is true, EOLN(FILEID) is also true and FILEID^ is undefined.

If EOF(FILEID) becomes true during a GET(FILEID) or a READ(FILEID, ..) then the data thereby obtained may not be valid.

As direct data files are randomly accessible, the MIKADOS system does not keep track of which records in a file have been accessed by the user. Consequently, MIKADOS does not have any information about the number of the last record accessed in a direct file. No EOF mark exists in direct files. EOF for a direct file becomes true only if an attempt is made to GET a record that lies partly or completely outside the file boundaries.

EOF on the INPUT file occurs when the operator terminates input by pressing ESC. An EOF condition on the INPUT file is reset when the next EDIT, or READLN on INPUT is encountered. The new READ or READLN may of course cause EOF to be set true again.

EOF on the OUTPUT file occurs when the operator enters a >.BR (break) command on the console that was used directly or indirectly to start the Pascal program. Note that the break condition is reset during the evaluation of EOF(OUTPUT). Consequently, EOF(OUTPUT) will be true only in the first evaluation after the >.BR command was entered.

7.16 EOLN - end-of-line

This function is similar to the EOLN function defined on page 160 of J&W. The following supplementary comments should be noted:

After a RESET or a REWRITE has been issued for a file, EOLN returns true. EOLN returns true on a closed file. The EOLN function should be used only on files of type TEXT or FILE OF CHAR.

7.17 Miscellaneous implementation size limits

The following is a list of maximum size limitations imposed upon the user by the current implementation of MIKADOS Pascal:

1) Maximum number of bytes of object code in a procedure or function is 2000. Local variables in a procedure or function can occupy a maximum of 16383 words of memory

2) Maximum number of characters in a string variable is 255

3) Maximum number of words in a set is 255;
   maximum number of elements in a set is 255 * 16 = 4080

4) Maximum number of segment procedures and segment functions is 15

5) Maximum number of procedures or functions within a segment is 127

6) Maximum number of external procedures is 32 minus number of module-dependent externaloptions (C, D, F) specified.

8. External procedures and functions

A user may reference a procedure (function) written in assembler or a similar language in a Pascal program by declaring it an external procedure (function). Except where otherwise noted the following discussion applies to functions as well as procedures.

An external procedure is declared by substituting in the procedure declaration an ´EXTERNAL´ directive for the procedure block, e.g.

```
PROCEDURE SORT( VAR A: ARRAYOFINTEGER; X: INTEGER ); EXTERNAL;
```

When compiling Pascal programs containing external references, the user must always set the appropriate externaloption (see section 4.1).

Note: names of external assembler procedures should not be longer than 5 characters as the assembler treats only the 5 first characters of an entry point name as significant while the Pascal compiler and the linker consider 8 significant characters.

8.1 Transfer of parameters

When a call to an external procedure is made in a Pascal program, the assembler routine is entered at the entry point corresponding to the procedure name.

On entry, the top of stack will contain the return address to the Pascal system. Next on the stack will be information about the parameters passed to the assembler routine in reverse order, i.e. last parameter first.

Variable (VAR) parameters are represented by a pointer to the actual structure referenced, while value parameters are either passed directly on the stack (unstructured parameters) or as pointers to temporary data structures.

Assembler functions pass their return values on the stack.

Assembler routines are responsible for removing all parameter information from the stack before returning control to the Pascal system.

Example:

Procedure P is declared as

```
PROCEDURE P( VAR P1: INTEGER;
                 P2: BOOLEAN;
                 P3: STRING ); EXTERNAL;
```

and is called by

```
P( I, FALSE, ´TESTING´ );
```

The Pascal system will analyse the parameters and call the assembler routine at entry point P (declared as P::). The stack contents on entry to P will be

top of stack	return address to Pascal system
next on stack	pointer to first byte (length indicator, =7) in copy of string constant in work area
third on stack	the value ´false´ (binary 0)
fourth on stack	pointer to the variable I
fifth on stack and	following should not be used by P

The routine P should remove the four topmost words from the stack and return to the specified return address after completing its task.

8.2 Parameter formats

Although an element of a structure may occupy as little as one bit as in a packed array of boolean, all parameters occupy at least one word (16 bits) even if not all the information in the word is valid. The least significant bit of a word is bit 0, the most significant is bit 15.

boolean: one word. Bit 0 indicates the value (false=0, true=1) and this is the only information used by boolean comparisons. However, the boolean operators AND, OR and NOT operate on all 16 bits.

integer: one word, two's complement, capable of representing values in the range -32768..32767.

scalar (user defined): one word, in range 0..32767

char: one word, with low byte containing character.

real: two words, with the following format

```
          ---------------------------------------------
word 1: !     low mantissa      !   middle mantissa    !
          ---------------------------------------------

          ---------------------------------------------
word 0: !sign! high mantissa    !       exponent       !
          ---------------------------------------------
```

The representation has an excess-128 base 2 exponent, a fractional mantissa that is always normalized, an implicit 24th mantissa bit, and zero represented by a zero exponent.

pointer: one or three words, depending on type of pointer.
Pascal pointers, internal word pointers: one word, containing a word address
Internal byte pointers: one word, containing a byte address
Internal packed field pointers: three words
word 2: word pointer to word field is in
word 1: field width (in bits)
word 0: right bit number of field

set: 0..255 words in data segment, 1..256 words on stack. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as SET OF M..N is allocated (N+15) DIV 16 words. When a set is in the data segment, all words allocated contain valid information. When a set is on the stack, it is represented by a word containing the length, and then that many words, all of which contain valid information. All elements past the last word of a set are assumed not to be elements of the set.

records and arrays: any number of words (up to 16384 words in one dimension). Arrays are stored in row-major order (last index varying most rapidly) and always have a lower index of zero. Packed arrays have an integral number of elements in each word as there is no packing across word boundaries (it is acceptable to have unused bits in each word).

strings: 1..128 words. Strings are a flexible version of packed arrays of characters. A string(n) occupies (n div 2)+1 words. Byte 0 of a string is the current length of the string, and bytes 1..length(string) contain valid characters.
A string constant of length 1 is considered a character constant unless the delimiter " is used instead of ´, i.e. in the assembler procedure call

```
ASM( ´y´, "y", ´yes´, "yes" )
```

the first argument will be transferred as a character constant (value on stack), and the other arguments will be transferred as string constants (pointer to string on stack).

8.3 Linking external procedures

After a Pascal program containing external references has been compiled, a new interpreter containing a copy of the external procedures referenced in the program must be linked.

Before linking a new interpreter, construct a special interpreter disc using the FCOPY program (described in the "MIKADOS Utility Programs and Subroutines" manual). The disc must contain the following:

- all MIKADOS system nullfiles
- a MIKADOS linker

Example: Pascal system disc in P1, user Pascal source disc in P2. Pascal compilation of "EXTDEMO":

```
>PASCAL,EXTDEMO,,E
```

The resulting P-code file and the resulting relocatable file will both be placed on P2.
Now insert the special interpreter disc in P1.
Linking of a new interpreter that will execute "EXTDEMO" assuming relocatable modules of external procedure(s) referenced are on either P1 or P2:

```
>LINK,EXTDEMO,R1
```

To execute EXTDEMO enter

```
>EXTDEMO
```

Note: do not enter >INTER,EXTDEMO

If a special interpreter for a program containing external references has previously been linked, it may be reused providing the order of declaration of the external procedures in the source program has not been changed.

9. Pascal E (Extended precision Pascal)

To satisfy users requiring high precision computations, e.g. for scientific or business applications, DDE has developed a special version of the Pascal compiler and interpreter called Pascal E.

In Pascal E real numbers are represented by 8 byte BCD coded strings. All computations are performed with 13 significant digits and an exponent range of 10**(-128) to 10**128 and true zero.

Any attempt to execute a Pascal program compiled by the extended precision compiler with a normal interpreter or vice versa may cause the system to crash or to behave erratically.

The following sections describe the differences between ordinary Pascal as described in chapters 1 - 8 of this manual and Pascal E.

9.1 Interpreter and compiler names

The name of the standard interpreter is INTRE (as opposed to INTER in standard Pascal). The name of the Pascal compiler is PASCALE (as opposed to PASCAL in standard Pascal).

9.2 Intrinsic functions

Pascal E supports the intrinsic functions SQRT, EXP, LN, TAN, SIN, COS and ARCTAN. These functions are not supported by normal Pascal.

9.3 Names of interpreter modules

The BCD floating point interpreter modules are

- FPE (basic BCD operations)
- FPEX (dummy version of FPE; used instead of FPE if BCD arithmetic is not required)
- FPBCD (basic BCD operations; may be omitted if BCD arithmetic is not required)
- FPT (BCD functions SQRT, EXP, LN, TAN, SIN, COS and ARCTAN)
- FPTX (dummy version of FPT; used instead of FPT if BCD functions are not required)

The FP and FPX modules used by normal Pascal should not be used with Pascal E. The FPEX and FPTX modules may be included in the interpreter instead of the above modules if no floating point operations are required.

The remaining interpreter modules are the same for PASCAL and PASCALE (see section 4.4), except that the central interpreter module is named INTRE instead of INTER.

9.4 Direct support of READREAL and WRITEREAL

The READREAL and WRITEREAL procedures are built into the interpreter. No inclusion is necessary to READ or WRITE real numbers.

A real number in the parameter list of a WRITE or WRITELN procedure call may be followed by :e1 or :e1:e2 as described in J&W (page 86), where e1 is the field width and e2 is the number of decimals printed.

e2 may be		
	>0	- e2 specifies number of decimals to be printed
	0	- use exponential notation for number
	-1	- no decimals printed, decimal point printed
	-2	- no decimals printed, no decimal point printed

If the values of e1 or e2 are unreasonable, the number is printed in exponential notation with e1=20.

9.5 Internal representation of real numbers

The real number format is

byte 0	bit 7:	sign (1 = minus)
	bit 3-0:	most significant BCD digit
byte 1-6	12 BCD digits	
byte 7	exponent expressed in excess 128 form	

The implied decimal point is located immediately to the left of the most significant BCD digit. True zero is represented by a zero exponent.

Example: the constant 1.56 is represented as

```
DB    01,56,00,00,00,00,00,81
```

Appendix A. Compile time error messages

Most of the error numbers are similar to those defined in appendix E of J&W. Errors with numbers > 400 cause the compiler to terminate. Only the first error detected on a source line is reported.

1: error in simple type
2: identifier expected
3: ´PROGRAM´ expected
4: ´)´ expected
5: ´:´ or ´..´ expected
6: symbol illegal in context (maybe missing ´;´ on the line above or ´;´ in front of ELSE)
7: error in parameter list
8: ´OF´ expected
9: ´(´ expected
10: error in type
11: ´(.´ expected
12: ´.)´ expected
13: ´END´ expected
14: ´;´ expected
15: integer expected
16: ´=´ expected
17: ´BEGIN´ expected
18: error in declaration part
19: error in <field list>
20: ´,´ expected
21: ´.´ expected

50: error in constant
51: ´:=´ expected
52: ´THEN´ expected
53: ´UNTIL´ expected
54: ´DO´ expected
55: ´TO´ or ´DOWNTO´ expected in FOR-statement
58: error in <factor> (bad expression)
59: error in variable

101: identifier declared twice
102: low bound exceeds high bound
103: identifier is not of the appropriate class
104: undeclared identifier
105: sign not allowed
106: number expected
107: incompatible subrange types
108: file not allowed here
109: type must not be real

110: <tagfield> type must be scalar or subrange
111: incompatible with <tagfield> part
113: index type must be a scalar or a subrange
114: base type must not be real
115: base type must be a scalar or a subrange
116: error in type of standard procedure parameter
117: unsatisfied forward reference
119: re-specified parameters not ok for a forward declared procedure
120: function result type must be scalar, subrange or pointer
121: file value parameter not allowed
122: a forward declared function´s result type cannot be re-specified
123: missing result type in function declaration
124: f-format for reals only
125: error in type of standard function parameter
126: number of parameters does not agree with declaration
127: illegal parameter substitution
128: result type does not agree with declaration
129: type conflict of operands
130: expression is not of set type
131: tests on equality allowed only
132: strict inclusion not allowed
133: file comparison not allowed
134: illegal type of operand(s)
135: type of operand must be boolean
136: set element type must be scalar or subrange
137: set element types must be compatible
138: type of variable is not array
139: index type is not compatible with the declaration
140: type of variable is not record
141: type of variable must be file or pointer
142: illegal parameter solution
143: illegal type of loop control variable
144: illegal type of expression
145: type conflict
146: assignment of files not allowed
147: label type incompatible with selecting expression
148: subrange bounds must be scalar
149: index type must not be integer
150: assignment to standard function is not allowed
152: no such field in this record
153: type error in read
154: actual parameter must be a variable
155: control variable cannot be formal or non-local
156: multidefined case label
157: too many cases in case statement
158: no such variant in this record
159: real or string tagfields not allowed
160: previous declaration was not forward
161: again forward declared
162: parameter size must be constant
163: missing variant in declaration

164: substitution of standard procedure/function not allowed
165: multidefined label
166: multideclared label
167: undeclared label
168: undefined label
169: error in base set
173: externaloption not specified in RUN command (see section 4.1)

201: error in real number - digit expected
202: string constant must not exceed source line
203: integer constant exceeds range

250: too many scopes of nested identifiers
251: too many nested procedures or functions
252: too many forward references of procedure entries
253: procedure too long
257: too many external procedures

320: READREAL or WRITEREAL procedure not included (see section 7.12 or 7.13)

398: implementation restriction
399: implementation restriction

400: illegal character in text
401: unexpected end of input
408: include control comment not allowed in inclusion file
10xx: error during open of inclusion file
11xx: error during open of source file
12xx: error during create/open of P-code file
13xx: error during create/open of relocatable file
14xx: error during output to relocatable file
15xx: error during output to P-code file
16xx: error during input from source file

In the error messages with numbers >= 1000 the last two digits represent the MIKADOS file system error code. The MIKADOS file system error codes are:

1: a file with the specified name does not exist
2: a file with the specified name already exists
3: no more room on disc
4: illegal record length
5: the file is being used by another user
6: the specified DCB is not open
8: attempt to extend a file more than 60 times
9: the file name is illegal (the first character is not printable)
10: illegal disc identification
11: attempt to position file to non-existent record
12: attempt to read or write a record that lies partly or completely outside the file boundaries

13: the file has not been opened for writing
14: the catalog on the disc is full
15: illegal DCB length
16: illegal number of sectors in file
17: illegal file type
18: the file name has not been reserved (returned by CLOSE if file
    not open or DCB bombed)
19: error in variable length record file format

40: disc drive not ready
42: hard error on disc
44: disc drive is write protected
48: illegal track/sector number or illegal buffer length
50: transfer extends past last sector of disc drive
52: illegal disc identification
69: no data area available to pass parameters to new process

-1: end-of-file disregarded
-2: input error (READ real)
-3: input error (READ integer)

MIKADOS file system errors 4, 15, 18, 48, 50, and 52 normally should not occur in Pascal systems. If they do the system has probably been destroyed by a pointer or index error.

Appendix B. Run time (interpreter) error messages

During interpretation of a Pascal program (including the Pascal compiler itself), the interpreter checks for a number of error conditions. If an error condition is detected it is reported with a message similar to the following:

RUN TIME ERROR x NEAR LINE yyyy

or

USER I/O ERROR zz NEAR LINE yyyy

where yyyy is the line number of the last executed Pascal statement that was compiled with the debug option enabled (see section 4.2). If the debug option was not enabled during any part of the compilation of the program section executed before the error occurred, yyyy will be 0000. x will be replaced by

D - division by zero
E - external procedure error (probably wrong interpreter used
    to execute program with external references)
I - invalid index (if index and range checking disabled only
    string index < 1 or > 255 detected)
K - stack or heap overflow
L - string too long or parameter error in intrinsic procedure
M - standard procedure not implemented
O - floating point overflow
P - floating point error (error in PWROFTEN call)
S - non-existent segment called (system error)
X - exitting procedure never called (system error)

If the Pascal compiler fails with an interpreter error message other than K, M, O, or P, start by correcting all syntax errors reported during the compilation. Then recompile the program. If the error persists, contact DDE.

If the Pascal compiler fails with interpreter error message K, increase the compiler data area (increase MAXADR, see section

4.4), or reduce the number of identifiers declared in the program.

If the Pascal compiler fails with interpreter error message M, then the interpreter used is not suitable for compiling the Pascal program (special compiler module or floating point module required in the interpreter).

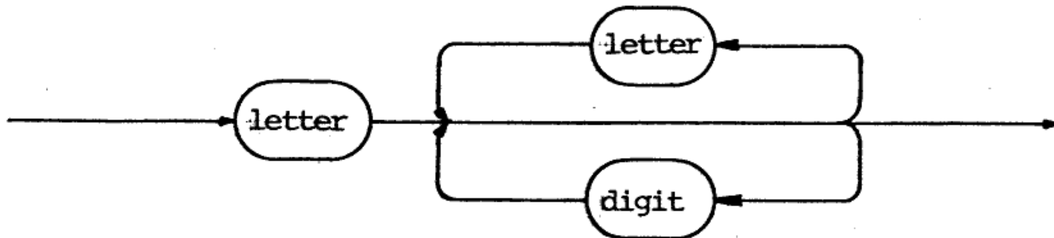
If the Pascal compiler fails with interpreter error message O or P, then the source statement just after the last one listed on the list device probably contains a (real) constant which is not acceptable to the compiler.

The user i/o error message appears if the result code of an i/o operation is greater than zero, and i/o checking has not been disabled (see section 5.2). zz is the result code which caused the error. The result codes (MIKADOS file system error codes) are explained in appendix A.

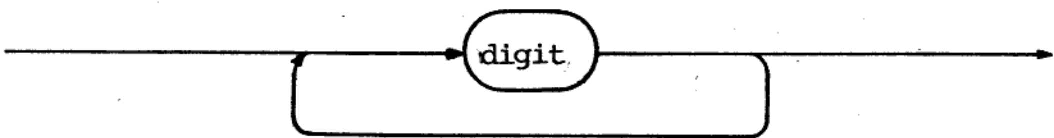
If a program (including the Pascal compiler) fails with an interpreter error, then the files opened by this program at the time when the error occurred are not closed. This may result in file system error 5 (file in use) in subsequent attempts to open the file. The error condition can be removed by restarting the operating system.

Appendix C. Revised syntax diagrams

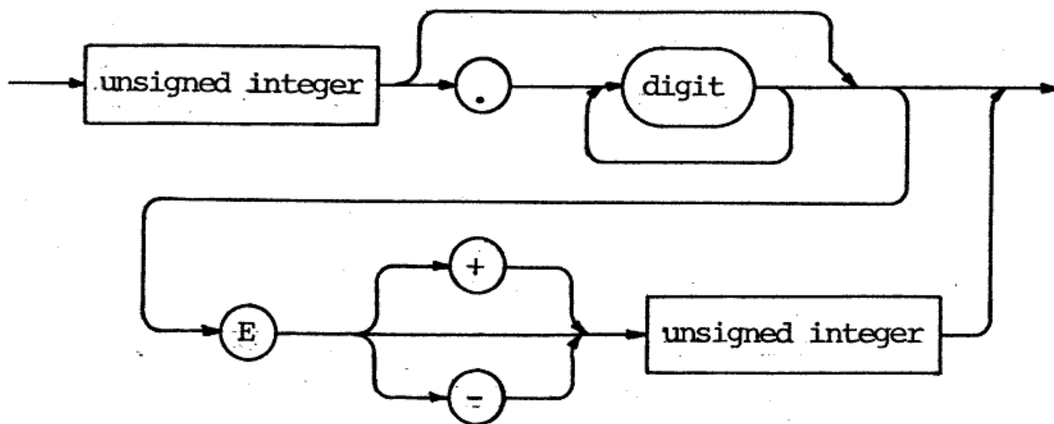
identifier



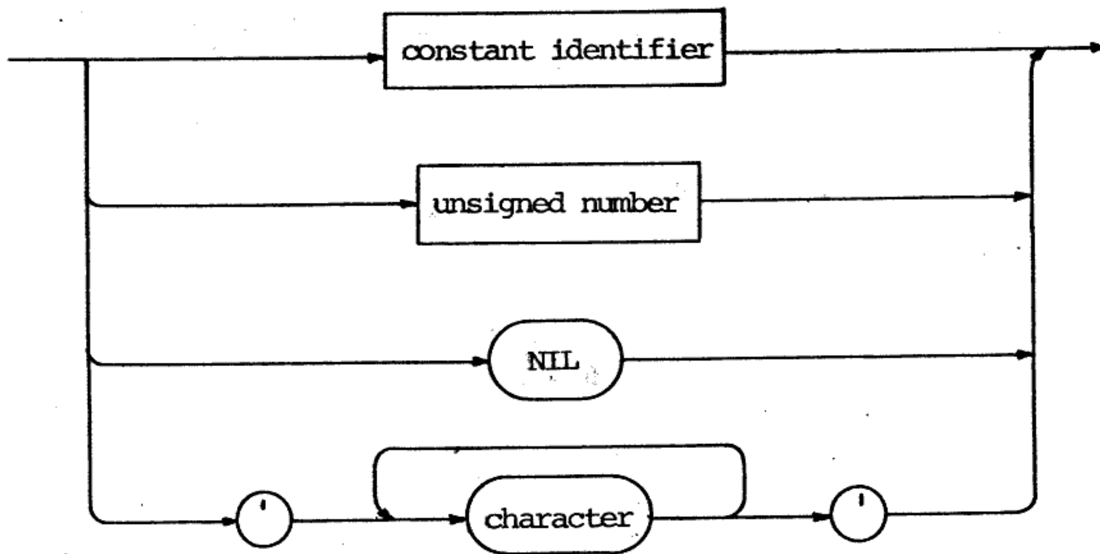
unsigned integer



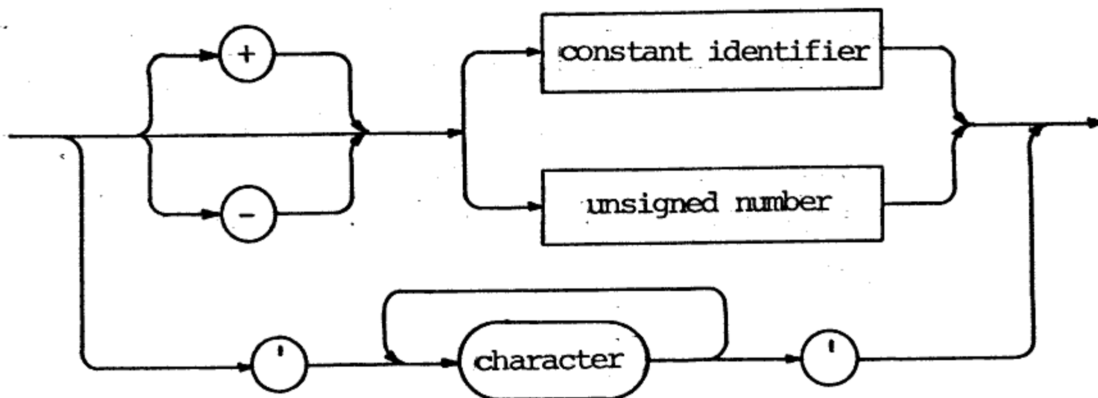
unsigned number



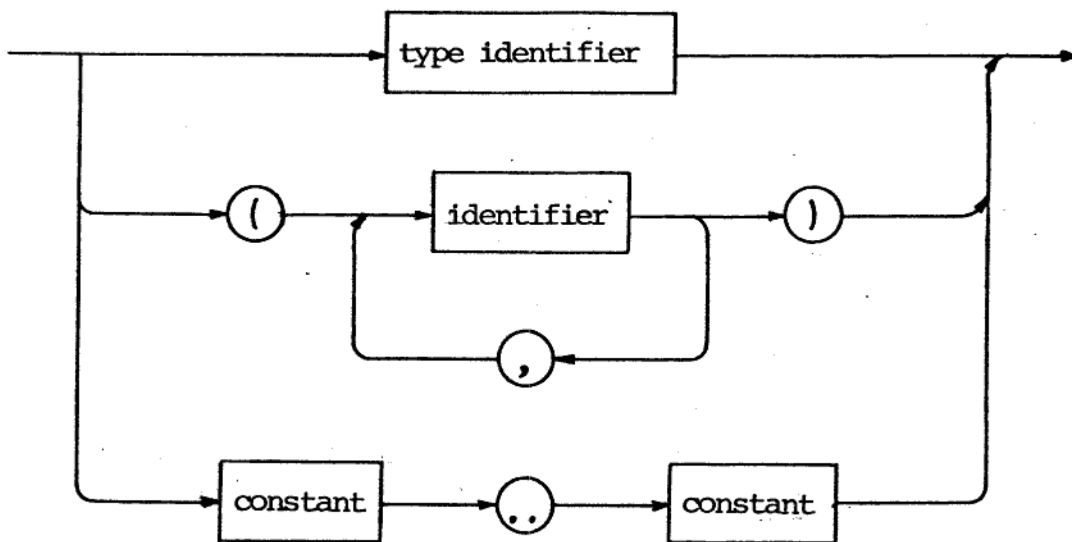
unsigned constant



constant



simple type



type

simple type

↑

type identifier

PACKED

ARRAY

(

(.

simple type

,

)

.)

OF

type

FILE

OF

type

SET

OF

simple type

RECORD

field list

END

STRING

(

(.

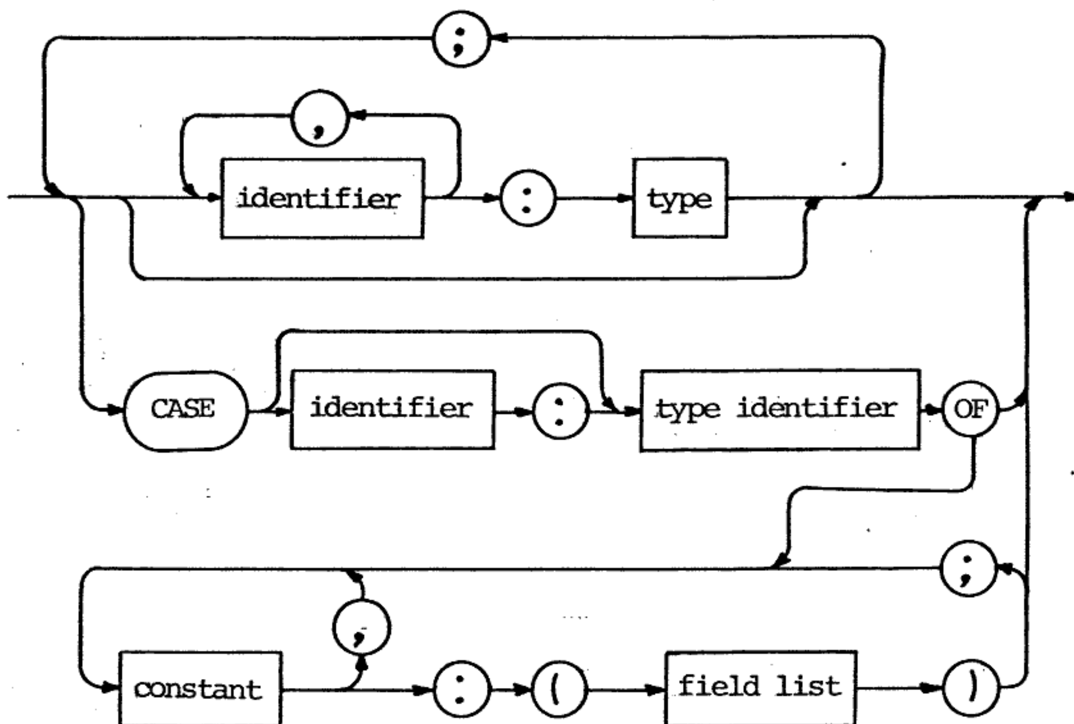
unsigned integer

)

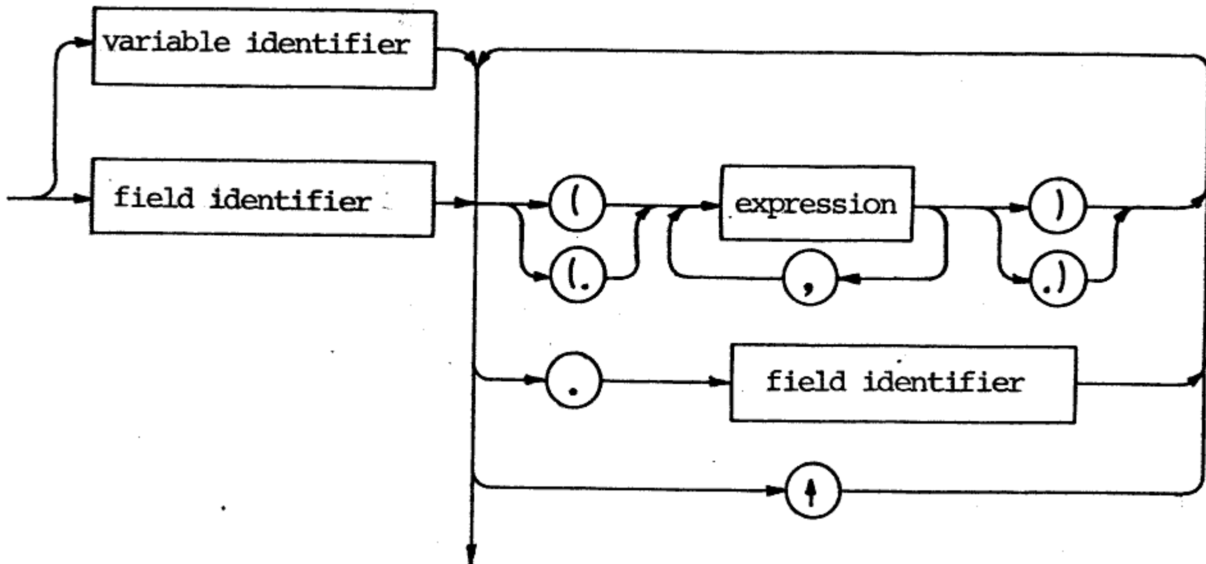
.)

TEXT

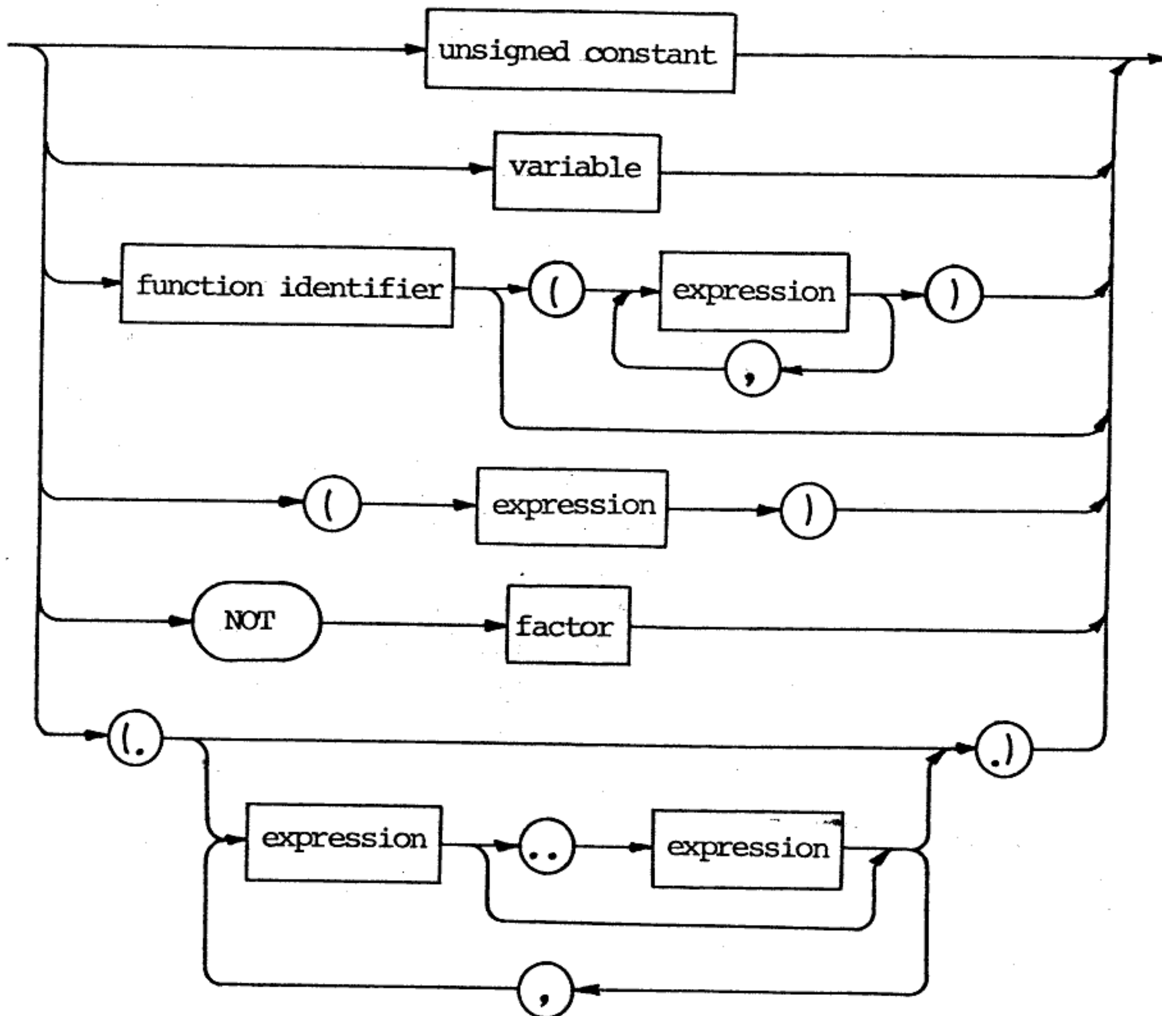
field list



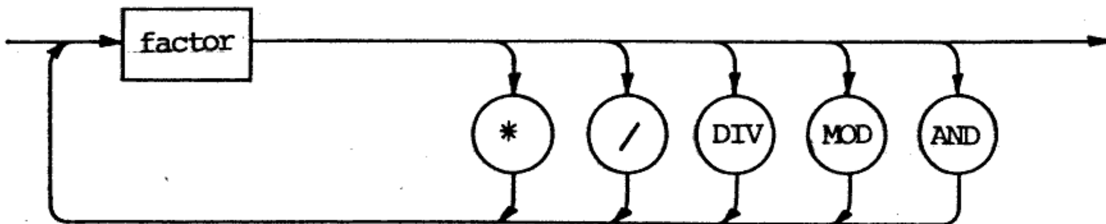
variable



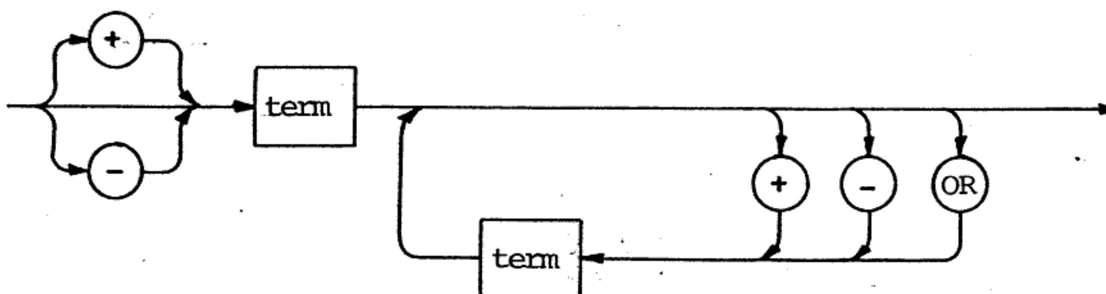
factor



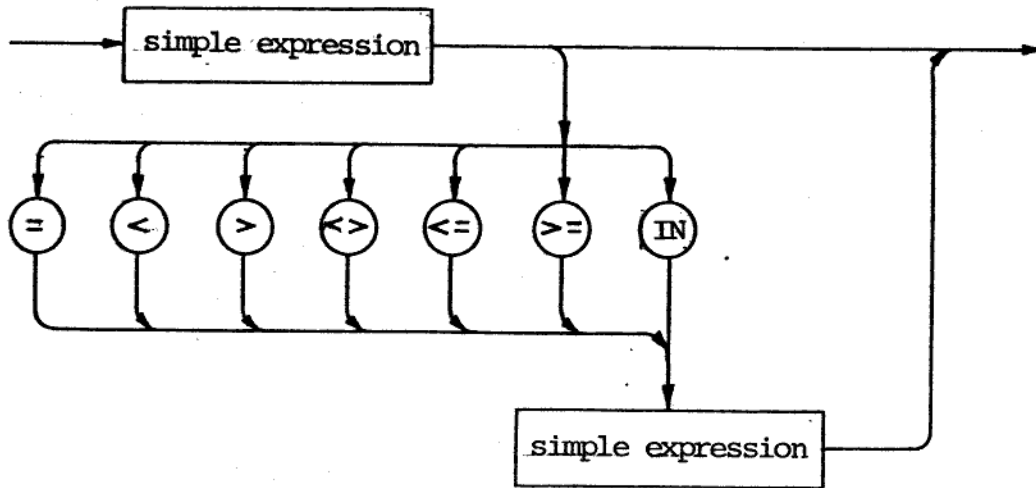
term



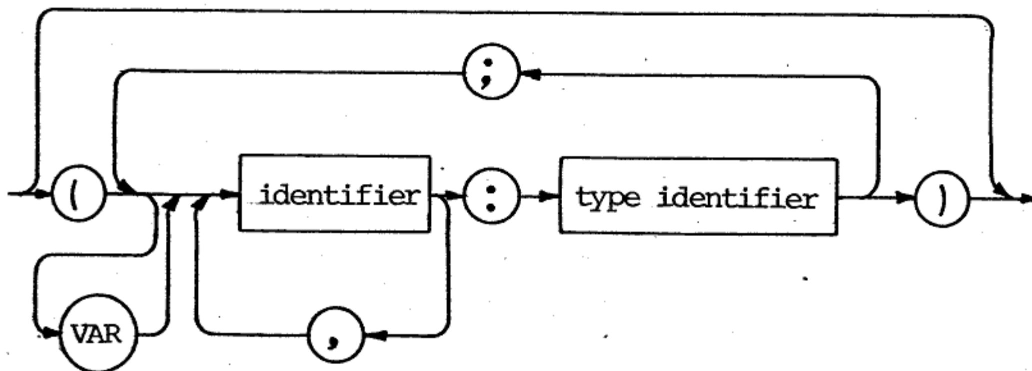
simple expression



expression



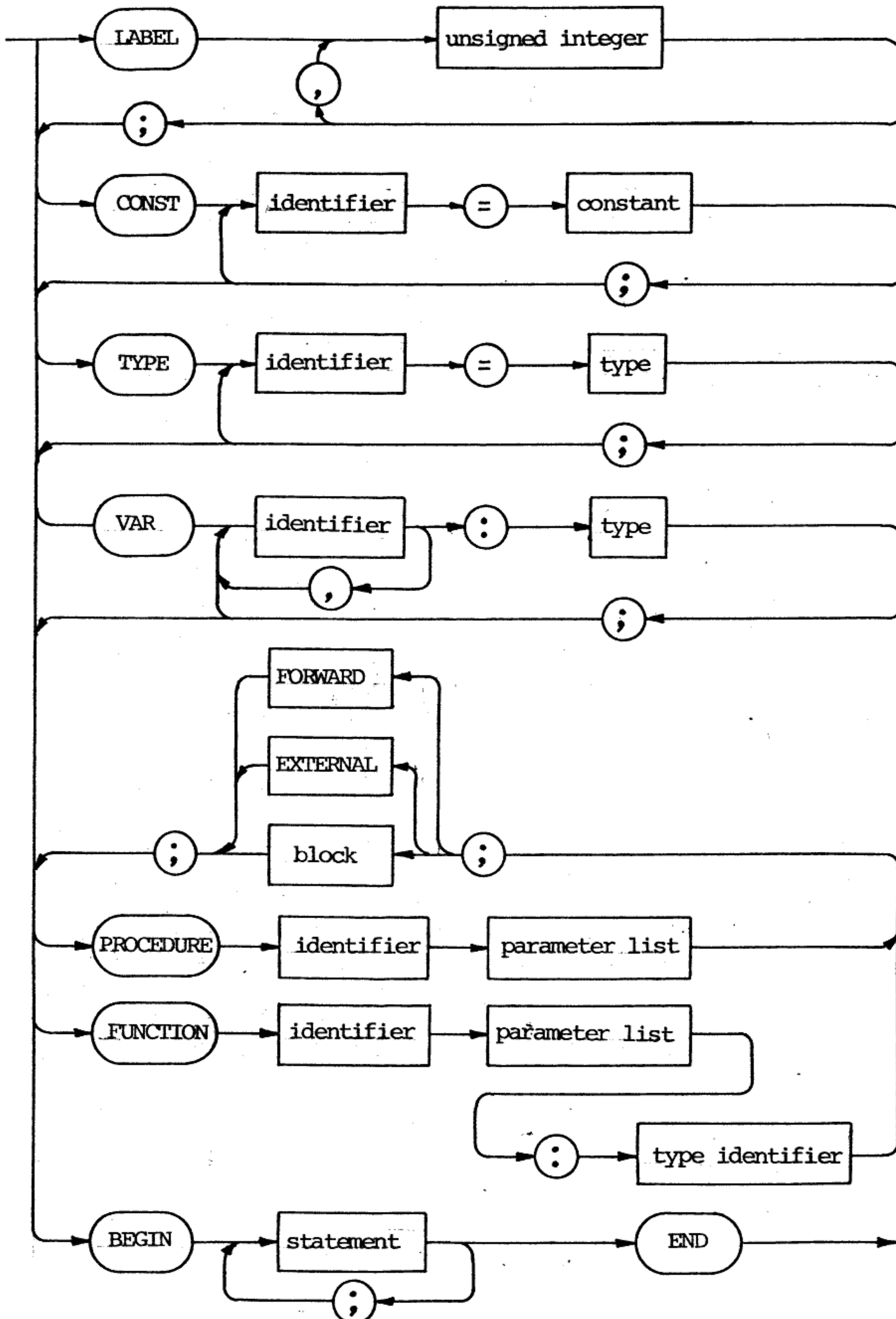
parameter list



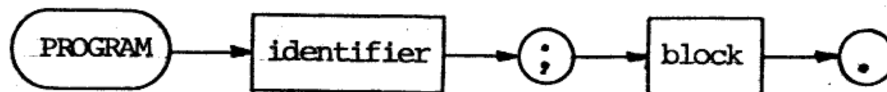
statement

unsigned integer
:
variable
function identifier
:=
expression
procedure identifier
(
expression
,
)
BEGIN
statement
;
END
IF
expres- sion
THEN
state- ment
ELSE
state- ment
CASE
expres- sion
OF
con- stant
,
:
state- ment
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
variable identifier
:=
expression
DOWNTO
TO
expression
DO
statement
WITH
variable
,
DO
statement
GOTO
unsigned integer

block



program



Appendix D. System performance data

D.1 Space requirements

The MIKADOS Pascal system consists of two separate programs, the P-code interpreter and the Pascal compiler.

The standard version of the P-code interpreter occupies approximately 9700 (hex 2600) bytes. A small, representative program (the HANOI program on the system demonstration disc) will run in a MIKADOS region of 12 k bytes (MAXADR=7000 if region start = 4000).

The minimum version of the interpreter (floating point, SEEK and special compiler module omitted) occupies approximately 8000 (hex 1F00) bytes. This interpreter requires a MIKADOS region of 10.5 k bytes or more to execute programs.

The Pascal compiler consists of the root segment, the initialization segment, which is called only at the start of a compilation, and the compilation segment. The approximate sizes of these segments are:

Root segment	100 bytes
Initialization segment	6900 bytes
Compilation segment	20800 bytes

To compile a small program (e.g. the HANOI program on the system demonstration disc) a region of at least 40000 bytes is required:

Minimum interpreter	8700 bytes
Compiler (longest segment)	21000 bytes
Compiler data area	10300 bytes

The compiler data area contains the compiler stack and heap.

Experience has shown that a region of 48 k bytes is sufficient to compile most programs. The Pascal compiler (a Pascal program of 4700 lines) requires 52 k to compile itself, using a special MIKADOS system where the region size is 53 k.

D.2 Execution times

The compiler compiles 3 - 5 lines of a "typical" Pascal source program per second. The Pascal compiler, which is a Pascal program of approximately 4700 lines with very few comments, compiles itself in approximately 19 minutes, giving a compilation speed of 4.1 lines per second on a 2 MHz CPU.

Execution times for typical Pascal constructs are:

```
FOR I:=1 TO 5 DO
  FOR J:=1 TO 10000 DO
    BEGIN                 47   seconds if debug option enabled
    END;                  38.5 seconds if debug option disabled
```

```
A := 5;              (A integer)                            140 us
A := B;              (A, B integer and local)               200 us
A := B + C;          (A, B, C integer and local)            370 us
E := 5.0;            (E real and local)                     400 us
E := F;              (E, F real and local)                  500 us
E := 5.0 + 5.0;      (E real and local)                    1080 us
D(4) := 5;           (D(1..5) integer and local)            510 us
IF TRUE THEN BEGIN END;                                      90 us
IF A=B THEN BEGIN END;    (A, B integer, local, A = B)      330 us
                          (same, A <> B)                    345 us
```

```
IF (H<´A´) OR (H>´Z´) THEN BEGIN END;
                          (H char and local, H = ´Q´)         650 us
                          (same, H = ´2´)                     620 us
                          (same, H = ´q´)                     620 us
IF NOT (H IN (.´A´..´Z´.)) THEN BEGIN END;
                          (H char and local, H = ´Q´)         800 us
                          (same, H = ´2´)                     770 us
                          (same, H = ´q´)                     770 us
TOMPROC;                  (call of empty procedure, level 0)  720 us
TOMPROC( A );             (call of empty procedure, level 0,
                           with one integer value parameter)  860 us
TOMPROC( A );             (call of empty procedure, level 0,
                           with one integer VAR parameter)    880 us
```

The above measurements were performed using the TIMING program on the Pascal demonstration disc with 50000 executions of two identical statements of the above types in a double FOR-loop as shown in the first construct above. All measurements were made on a 2 MHz CPU.

Appendix E. Summary of manual changes

The following is a summary of the changes that have occurred in this manual:

4 May 1979 original version

27 July 1979 section 2: contents of Pascal system disc slightly changed
- 4.0: description of MIKADOS IN command removed
- 4.2: P control comment (page eject) added
- 4.4: standard interpreter configuration changed
- 5.2 and 5.2.4: support for soft error codes (IORESULT<0) added
- 5.2.2: automatic CLOSEing of files
- 5.2.9: EDIT procedure added
- 5.2.10: CLEARSCREEN procedure added
- 5.4.2: TIME function unit changed
- 5.4.6: Basic i/o routines added
- 5.4.7: SETIORESULT procedure added
- 5.4.8: DELAY program procedure added
- 7.8: description of standard types added
- 7.10: relation between upper and lower case characters explained
- 7.12: IORESULT error codes generated by syntax errors in input
- 7.14: EOF condition on standard INPUT file
- 9: description of Pascal E added

4 Dec 1979 section 4.1: T listoption added (p-code size)
- 4.1: compiler break facility added
- 4.4: SEEK module renamed to FSYS; dynamic data area acquisition for interpreter added (MAXADR = 0)
- 5.2.8: use of SEEK in update operation clarified

5.4.9:	start new process procedure added (CHAIN)
7.15:	interpreter break facility added
7.15:	use of EOF with direct access files clarified
8.0:	restriction on external procedure name length documented
9:	rewritten

Description of changes in the Pascal system since December 4, 1979 (last edition of Pascal manual).

1) When the listoption T is given to the Pascal compiler it will output after each record in the type definition part the size of the record (number of bytes).

2) The compile time option N makes it possible to change the number of lines per page in a program listing from the Pascal compiler. The integer after N determines the number of lines per page. The integer must occupy two positions. Example: (*$N30*)

3) It is now possible to read and set the MIKADOS date and time from a Pascal program. The user must make the following declarations:

```
TYPE PARMARRAY = PACKED ARRAY (1..40) OF CHAR;
     CLOCKRECORD = RECORD
                     DATE: PACKED ARRAY (1..10) OF CHAR;
                     TIME: PACKED ARRAY (1..8)  OF CHAR
                   END;

VAR PARM: ^PARMARRAY;
    CLOCK: ^CLOCKARRAY;
```

The VAR declaration must be the first VAR declaration made in the program. After startup CLOCK^.DATE(1..10) will contain the date in the form DD.MM.YYYY. CLOCK^.TIME(1..8) will contain the time in the form HH.MM.SS.

4) In the procedure CHAIN a new parameter has been added.

```
PROCEDURE CHAIN(PROGRAMNAME, PARMSTRING: STRING;
                                    VAR ADDRESS: ^INTEGER);
```

After execution of CHAIN the new parameter contains a pointer to the process control block of the initiated process. The new parameter is intended for use by future application programs.

5) After the RESET and REWRITE procedure calls no i/o check instruction is generated.

6) It is not allowed to use the following form of the RESET procedure call (cfr. manual page 5.6):

```
PROCEDURE RESET(FILEID: PHYLE);
```

7) If the size of the primary file extent in a REWRITE procedure call is specified as zero or omitted then the file is not created if it does not exist.

8) The maximum size of a textfile record has been increased from 80 to 136 characters.

9) In MIKADOS systems with a version date of 13.03.1980 or later the record size in a direct file must not exceed 464 bytes. In older MIKADOS systems the record size in a direct file must not exceed 255 bytes.

10) Before an assembler function places its return value on the stack it must remove from the stack a number of words equal to the size of a real variable (4 in Pascal E, 2 in normal Pascal). A boolean assembler function called from a Pascal E program has to do as the following example shows if the return value is TRUE:

```
MVI     C,1
POP     H         ; return address
POP     D         ; a real occupies 4 words
POP     D         ; in Pascal E
POP     D
POP     D
PUSH    B         ; place TRUE return value on
                  ; stack
PCHL              ; return to interpreter
```

It is assumed that the assembler function does not have any parameters. The parameter transfer mechanism used by functions is identical to that used by procedures.

The following compile time error messages have been added:

256: CASE label range too long.
403: `PROGRAM` expected.

The following run time error messages have been added:

A - illegal element in OPEN/CLOSE table (system error)
F - attempt to open more then 10 files simultaneously
T - attempt to write more then 136 characters in a textfile record