Marinchip 9900 Pascal User Guide

Table of contents

1. Introduction -2
   1.1. Acknowledgements -2

2. Pascal references -2

3. Using Pascal -3
   3.1. Writing the program -3
   3.2. Compiling the program -3
   3.3. Executing the program -4
   3.3.1. Program parameters -4
   3.4. File compatibility -5
   3.5. Compiler temporary file usage -5

4. Changes to the Sequential Pascal language -5
   4.1. Scanner changes -5
   4.1.1. Upper and lower case -5
   4.1.2. Standard operator characters -5
   4.1.3. Comment delimiters -6
   4.1.4. Odd length strings -6

5. Error messages -6
   5.1. Run-time errors -6
   5.1.1. Overflow -6
   5.1.2. Pointer error -6
   5.1.3. Case or subscript out of range -6
   5.1.4. Tag field incorrect for variant reference -6
   5.1.5. Heap overflow -7
   5.1.6. Stack overflow -7
   5.2. System error messages -7
   5.2.1. Bad program name -7
   5.2.2. Huh? -7
   5.2.3. Bad parameter -7
   5.2.4. I/O error -7
   5.2.5. I/O error on file <filename> -7
   5.2.6. Kernel called by sequential program -7
   5.3. Compiler error messages -7
   5.3.1. Try again -8
   5.3.2. Temporary file missing -8
   5.3.3. Object file lost -8
   5.3.4. Source file unknown -8
   5.3.5. Destination file unknown -8
   5.3.6. Compilation errors -8

1. Introduction

Marinchip 9900 Pascal is an adaptation of the programming language Sequential Pascal for the Marinchip 9900 computer system. Sequential Pascal is a minicomputer-based version of the programming language Pascal, which is widely regarded as the leading language in use today for the development of reliable software through the techniques of structured programming. Pascal contains the features found in other programming languages such as BASIC, Fortran, and Algol, but adds the following important capabilities:

Pascal allows the user to define new data types, and to thereby extend the language. The error-prone coding of program items as explicit numbers is eliminated.

Pascal allows structures of data to be built. This allows the definition and convenient use of tables containing various kinds of data that are essential in software development.

Pascal has extensive compile-time checking of program correctness. The compiler detects all inconsistencies in use of variable types, which catches most of the frequently-made errors in Fortran, BASIC, or Assembly language programming.

Pascal provides pointer variables and dynamic storage allocation. These features permit virtually all systems programming (e.g., compiler and operating system) tasks to be written entirely in Pascal.

Pascal provides powerful program control statements which eliminate the need for the "go to" statement. Programs written without "go to" statements are much easier to understand and debug.

Marinchip Systems has integrated Pascal into the Marinchip Disc Executive, so that Pascal programs may be called like any other program within the system. The Pascal compiler is a Pascal program that runs under the control of the Disc Executive.

1.1. Acknowledgements

The Marinchip Pascal system is based upon the Sequential Pascal compiler designed by Per Brinch Hansen and implemented by Alfred C. Hartmann on a PDP 11/45 at the California Institute of Technology. The design of the Pascal runtime system which interfaces Pascal programs to the Marinchip Disc Executive draws upon the design of the Solo operating system, designed and implemented by Per Brinch Hansen, and the Concurrent Pascal interpreter and kernel, written by Robert Deverill and Tom Zepko.

2. Pascal references

This manual will not attempt to teach the language Pascal. The user is referred to the following references for a description of the language. The references will be cited by the numbers given throughout the text of this manual.

[1] Brinch Hansen, Per, The Architecture of Concurrent Programs. Prentice Hall, Englewood Cliffs, New Jersey, 1977. This book describes the Solo operating system and the Concurrent Pascal language. The Solo operating system environment is simulated by Marinchip Pascal, so this book is the definitive reference on user-level programming in Sequential Pascal. The information on the language Concurrent Pascal itself is applicable to Marinchip Concurrent Pascal, another compiler available for the M9900. This is also one of the best books about developing operating system software and the design of large systems in general that has ever been written.

[2] Brinch Hansen, Per, and Hartmann, Alfred C., Sequential Pascal Report. California Institute of Technology, Information Science, 1975. This document is the formal definition of Sequential Pascal and describes

the language as it exists on the M9900. It is written to be used as a reference to the language, not a tutorial in Pascal. It is indispensable for any serious user of Pascal.

[3] Jensen, K., and Wirth, N., Pascal - User Manual and Report (Second edition). Springer-Verlag, New York, 1974. This is THE tutorial manual for Pascal. It describes a Pascal compiler for the Control Data 6600 which differs in some respects from the Sequential Pascal compiler for the M9900, but still is an excellent introduction to Pascal for users familiar with other programming languages as well as those learning programming in Pascal.

Finally, if like most users of Pascal, you become a Pascal fanatic, exhorting all listeners on the wonders of writing programs that work the first time and continue working forever, you should join the Pascal User's Group, which publishes a quarterly newsletter containing all the news about Pascal from around the world. This is the major forum for reporting implementations, experience using implementations, and commentary regarding the language, its use, and proposed modifications. To join, send $4 for one year of membership to:

Pascal User's Group, c/o Andy Mickel
University Computer Center: 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455

3. Using Pascal

This chapter describes how to write, compile, and run Pascal programs on the M9900 system. The user is assumed to be familiar with use of the Marinchip Disc Executive and the Text Editor. The reader who is unfamiliar with those system components is referred to the respective User Guides for more information regarding them.

3.1. Writing the program

A Pascal program is written according to the language specifications in the references given in chapter 2, and entered into the system with the Text Editor. Since input to the Pascal compiler is totally free-form, the user need not be concerned with column alignments when entering the data. Any sequential Pascal program begins with a "prefix" which defines the interface between the program and the operating system. This prefix must be added to a program before it is compiled. This is most easily accomplished by the "RF" command in the text editor. The user should go to the top of the file, and enter the command:

RF PREFIX

to the text editor, where PREFIX is the name of the file on the system disc containing the standard prefix. (The standard prefix is supplied by Marinchip Systems in a file by that name). This command will copy the prefix before the first line of the user program.

The user is required to explicitly add the prefix to programs because with the Concurrent Pascal compiler, the user can define new operating systems having different interfaces to programs. In order to allow Pascal programs to be run under these new systems, the ability to change the prefix must be available. A complete definition of the standard prefix explaining it line by line may be found in section 5.2 of reference [1].

3.2. Compiling the program

Once the Pascal program has been stored in a file with the Text Editor, it may be compiled with the Pascal Compiler. The compiler is called from the command level of the operating system by the statement:

PASCAL(<input>,<listing>,<object>)

where <input> is the file containing the Pascal source program, <listing> is the file where the compilation listing should be placed, and <object> is the file in which the executable object code will be written by the compiler. The <listing> file may be either a disc file or a device file such as the console or a printer. If the listing file specification is omitted (only two commas appear between the <input> and <object> file names), no program listing will be generated. Error messages, if any, will still be printed on the system console. For example, to compile a program named CALC, placing the executable code in a file named XCALC, and sending the compile listing to the console, the command would be:

```
PASCAL(CALC,CONS.DEV,XCALC)
```

If no listing were desired, the command would be:

```
PASCAL(CALC,,XCALC)
```

If the compilation is successful, the compiler will simply exit back to the operating system and the operating system command prompt will reappear. If the compiler found errors in the program, the message:

```
Compilation errors
```

will appear on the console. If the compilation listing has been sent to a disc file, it must be examined to determine the cause of the errors.

3.3. Executing the program

Once a Pascal program has been successfully compiled, it can be executed like any other program under the Disc Executive, simply by typing the name of the file containing the program. Note that it is not necessary to link Pascal programs before execution: they are directly executable after compilation. To execute the program XCALC compiled above, one would simply type:

```
XCALC
```

on the console when the operating system command prompt appears.

3.3.1. Program parameters

Programs called from the console may be passed parameters. These parameters are enclosed in parentheses following the name of the file containing the program. The file names passed to the Pascal compiler are an example of program parameters. Program parameters may be either integers (simple numbers), Boolean values (represented by the words TRUE or FALSE), or identifiers (any non-numeric character string other than TRUE or FALSE). If two consecutive commas appear, the parameter in that position will be considered to be omitted, and its type will be set to NILTYPE. Parameters are separated by commas, just like procedure parameters in Pascal. The structure of program parameters and their use within a Pascal program is explained in the section "Program Parameters" in chapter 5.2 of reference [1]. To ease compatibility between Pascal programs and programs written in other languages, the parentheses surrounding the parameter list are optional. Hence, the two program calls:

```
        PROG1(OUT,IN,10)
and:    PROG OUT,IN,10
```

are equivalent.

Pascal programs are allowed to call other Pascal programs in the M9900 implementation, and programs can pass parameters to the programs t in the same manner as parameters are entered on the console. This allows Pascal programs to be called either from another program or directly from the console without any modification of the program. It also allows Pascal to be used directly as the system command language, avoiding the need for a special job control language.

3.4. File compatibility

Marinchip Pascal reads and writes ASCII files that are compatible with those used by all other Marinchip software. The standard WRITEARG, READARG, READ, and WRITE procedures of Sequential Pascal are used to read and write files. Both disc and device files may be used without problems. The capability to use another Pascal program as input or output from a program is not currently provided in Marinchip Pascal: input and output must be files. The READPAGE and WRITEPAGE I/O procedures are also supported as alternatives to READ and WRITE.

The OPEN, CLOSE, PUT, and GET I/O mechanism of Sequential Pascal may be used to read and write any disc file. The random-access nature of these requests will be ignored if they are used on device files.

The ACCEPT and DISPLAY procedures are implemented according to the Sequential Pascal documentation for communication with the system console.

The LOOKUP procedure accesses the system file directory. The identifier passed to LOOKUP may be a fully general file name as described in the Disc Executive User Guide.

3.5. Compiler temporary file usage

The Pascal compiler uses the two system standard temporary files, TEMP1$ and TEMP2$, during the compilation. If for some reason other files are to be used (for example, when compiling a very large program where the standard temporary files are not large enough), this may be accomplished by specifying the temporary files in the call on the compiler:

PASCAL(<input>,<listing>,<object>,<temp1>,<temp2>)

If called in this manner, the compiler will use the named files for its temporary files.

4. Changes to the Sequential Pascal language

This chapter describes the modifications made to the Sequential Pascal language by Marinchip Systems. Programs written with the intention of being run on other machines offering Sequential Pascal should avoid the use of the features added in the Marinchip implementation.

4.1. Scanner changes

The scanner phase of the compiler has been modified to permit optional use of a syntax more like that of standard Pascal. The Sequential Pascal syntax is still accepted without modification.

4.1.1. Upper and lower case

Identifiers and reserved words are now accepted in both upper and lower case. The case used in a letter has no effect on matching, hence the following identifiers are considered identical by the compiler:

ARGLEBARGLE arglebarge ArgleBargle ArGlEbArGlE

4.1.2. Standard operator characters

Sequential Pascal was originally developed on a card system, so several of the standard Pascal operators that do not appear in the Hollerith code were redefined. Marinchip Pascal accepts both the standard character as well as the Sequential Pascal circumlocution.

Sequential Pascal	Standard Alternate

4.1.3. Comment delimiters

Sequential Pascal normally uses the double quote character (") to delimit comments. Marinchip Pascal also allows the standard Pascal comment delimiters { and } to be used. The curly brackets ({}) will be ignored in a comment bracketed by double quotes ("), and double quotes will be ignored within a comment defined with curly brackets. This allows a limited nesting of comments, which may prove useful when it is desired to turn off code containing comments.

4.1.4. Odd length strings

Sequential Pascal normally requires that all constant strings (other than single characters) contain an even number of characters. Since this forces the user to laboriously count characters unnecessarily, this restriction has been removed in the Marinchip compiler. Strings which contain an odd number of characters will be padded with an ASCII NUL '(:0:)' character to extend them to an even number of characters. This has no impact on existing programs, because such strings would have caused compile errors in previous compilers.

5. Error messages

This chapter describes error messages that may appear in the process of using the Pascal system.

5.1. Run-time errors

The following messages indicate an error detected during execution of a Pascal program. All of these messages will be followed by a line which identifies the source line number within the Pascal program where the error occurred.

5.1.1. Overflow

An arithmetic overflow was detected. Integers are limited to the range -32768 to +32767, and reals are limited to approximately +-10E75.

5.1.2. Pointer error

An attempt was made to use a pointer which did not point to anything.

5.1.3. Case or subscript out of range

Either a subscript was outside the bounds of the array it was used to subscript, or the value of the expression in a CASE statement failed to match any of the case labels in the statement.

5.1.4. Tag field incorrect for variant reference

An attempt was made to use a field in a variant record when the tag field indicated the record contained a different structure than the one used.

5.1.5. Heap overflow

The program has run out of storage when allocating new data items with the NEW construct. Check for runaway allocation or bad storage management.

5.1.6. Stack overflow

The program has run out of execution stack. Check for runaway procedure recursion.

5.2. System error messages

The following messages are generated by the Pascal system. They may occur either in the process of compilation or execution.

5.2.1. Bad program name

The program name given to the system is badly formed.

5.2.2. Huh?

The program name given to the system cannot be found on the disc.

5.2.3. Bad parameter

The program parameters were bad. The parameters must be separated by commas, and consist only of simple integers, the Boolean constants TRUE and FALSE, or identifiers of 12 characters or less.

5.2.4. I/O error

An unrecoverable I/O error has occurred either loading a Pascal program, or servicing an I/O request made by a program.

5.2.5. I/O error on file <filename>

An I/O error has occurred when using the GET or PUT I/O mechanism on the named file. This message may appear during a compilation. If the named <filename> is the object file specified on the call to the compiler, the object file is probably too small to hold the generated object code. If the <filename> is TEMP1$ or TEMP2$, larger temporary files are required to compile the program. See the section "Compiler temporary file usage" above for information on how to specify alternate temporary files for the compilation.

5.2.6. Kernel called by sequential program

This message indicates an internal error in the Pascal system. It may be caused if an object file created by the compiler is overwritten before execution.

5.3. Compiler error messages

The following messages are typed by the compiler on the system console when errors occur. These are not the messages which indicate source program errors which are placed in the listing file. Those messages are

generally self-explanatory.

5.3.1. Try again

This message is given when the compiler is called with bad parameters. A sample call on the compiler will be typed following this message indicating the proper form of a call on the compiler.

5.3.2. Temporary file missing

The compiler requires two temporary files, TEMP1$ and TEMP2$, to be present on the system disc. If either or both of these files are missing, this message will appear.

5.3.3. Object file lost

The compiler was unable to save the object code in the named object file.

5.3.4. Source file unknown

The source input file specified does not exist.

5.3.5. Destination file unknown

The listing file named does not exist.

5.3.6. Compilation errors

The program compiled contained errors. The user should examine the listing file for compiler error messages.

Documentary support for Mdex Pascal is very sparse, or horrendously expensive. Brinch Hansen's book [ref.1.], cost me almost $60, but it is not very helpful. These notes are intended to help in getting started. They should be read alongside the User Manual and a print-out of the Prefix from the master disc. The program Copy included works(!) but was written more to illustrate points raised than to have any enduring usefulness.

The Pascal Program.

This must take the form:-
Prefix, followed by Definitions & Declarations of
Constants, Data Types, Variables and Routines, (Procedures & Functions).
These are followed by the sequence of Statements forming the main program which will be executed one at a time.

The program is automatically stored on disc after compilation, and is activated by typing its name and any parameters on the console. To be pedantic, it is the name of the file containing the program which is typed. It is permitted, but surely confusing, to use another for the program.

The program Copy would be called:-

```
Copy (This_file,That_file)
```

File names must comply with Pascal identifier syntax, but may be standard Mdex names. The drive number on disc files is not taken as part of the Pascal identifier. Device files are named as per Mdex, i.e. Cons.dev, Print.dev, etc. This causes some restrictions, see later.

Each parameter in the list, more correctly called the Argument list, may be referenced by number. The system always inserts an extra param of type boolean which may be used to vet program performance. This extra one is Param[1] (or Param(.1.)); so This_file is Param[2], and so on. Ten arguments are allowed, any not mentioned are set to Niltype. See the record type Argtype in Prefix.

The Prefix.

Used by the authors of the language to hive off all hardware related facilities to ease transportability between different implementations. It consists of dummy routines which define the syntax of calls upon them, and direct execution to the actual routines within the operating system. Some routines must be called before others become effective.

File manipulation.

There are two distinct groups for this, they serve to tell the system the names and intended use of the files.

1. READ/WRITE.
   A call on the Prefix procedure Writearg activates this group:-
   Writearg(inp,source) or Writearg(out, dest) opens
   the files named in the variables Source and Dest
   which must be defined as Argtype.
   This activates the procedures Read, Write, Readpage
   and Writepage which may then be used to access
   their files sequentially from top to eof.
   The closure call is Readarg(inp, source) or Readarg(out, dest).
   They return a boolean to inform on satisfactory
   file usage.
2. Open(fileno,filename,found) associates an integer
   with the file by which it is thereafter referenced.
   This can only take the values 1 or 2, so only
   two files can be open at a time.
   This enables Get, Put and Length. Get & Put can
   set the file pointer to any page so give random access.
   Close terminates access and allows re-allocation of the file number.

Console I/O.

If the console is described to the system as Cons.dev, all the above file routines are applicable. The random nature of Get/Put is ignored. However, the console may be addressed by calls to two special routines, Display and Accept. These transfer single Ascii chars and need no prior activation. (see Procedure Context in Copy).

I/O Ascii restriction.

Pascal can only accept ascii chars so all other types will need conversion. Pasedit on the master disc provides much of what is needed for the console. If its small demo program is deleted, it can be copied to a user program in place of Prefix since that is already present. User's global Constants, vars,etc., should be placed between the Prefix and Pasedit. Predefined conversion functions are:-

ORD (x) The ordinal value of the char (x). (its ascii code)
CHR (x) The char whose code is (x).
CONV (x) The Real corresponding to integer (x).
Trunc (x) The Integer corresponding to the real (x).

The Lookup procedure locates a nominated file in the directory and if found returns a boolean True, and the Attributes of the file (see Prefix type Fileattr & Filekind).

Identify tells the system the program name. This is printed on the console before anything else. It is useful as a de-bugging aid, or to indicate the source of results when other programs are called from one running. It is printed once only for each phase. It is optional and is the only use of the pgm name during execution.

Run calls a program from one running. The name and params are exactly as would be typed for a console call, but two extra variables must be supplied as params. They are used after return to the caller, to show Where and How the called pgm was teminated.

Xpeek/Xpoke similar to Basic Peek/Poke.

Xcall calls an Assembler program into action from a running Pascal pgm. (I have not used this yet but it seems simple!

All routines starting IO....

A problem here as most use Prefix descriptions of peripherals such as Typedevice, Printdevice, etc. I have never had these accepted by the compiler which only recognises Mdex Cons.dev, Print.dev.etc. Such routines thus do not work. Exceptions are the types Ioresult and Ioparam. Both of these are effective.

Taskkind. Three tasks are defined, Inputtask, Jobtask & Outputtask. I have never been able to get out of Jobtask. Brinch Hansen says that Readline/Writeline are not effective in Jobtask. They don't work.

Program p(var param:arglist)

This is called when the pgm is activated from the console. No other definition Program must appear.

Other Features. Pre-defined operators for:-

Sets	OR	Union	= Equal
	&	Intersection	<> Not equal
	-	Difference	<= Contained in
	IN	Membership	>= Contains
	:=	Assignment	

Enumeration types
:= < = > <= <> >=
These have the expected meaning. Results are always Boolean.

Boolean & OR NOT where results are either False or True.

Integer + - * DIV MOD

Real = < > <> <= >= + - * / the pre-defined function ABS(x).

Arrays := = <>

Strings < > <= >=

Records := = <>

Storage requirements in bytes
Enums:2; Reals:8; Sets:16; String of m chars:m;.

Control Chars. These must be written in ascii coded form. e.g. '(:10:)' is the form for Line Feed. They are string vars.

Some Standard Functions. For enumerations,
SUCC (x) and PREC (x) return the successor or predecessor value of x (if there is one).

Arithmetic Functions.
Brinch Hansen Ref[1] is silent on this topic. Some which are implemented are:-
ABS(x), SQR(x) square of x, SIN(X).
these return a result of the same type as x , but that must be Real or Integer.
COS(x), ARCTAN(x), LN(x) the natural log.
These must all be Reals.
There must be others, perhaps somebody knows?

Variable & Universal Parameters.
When a procedure is given parameters, it may normally use the value supplied but is not allowed to change it. These are Constant Params. If the procedure is required to change the value, the param identifier must be preceeded by the symbol VAR. e.g. Procedure Something (addr:integer; var block:page)
Of the two params the routine may change the value of 'block', but not 'addr'. Similarly, it cannot always be known in advance what type of data a variable will be called upon to accept. This may be accomodated by writing UNIV in front of the identifier.
e.g. Procedure Something (univ text:line).
This makes 'text' a universal param which can accept any data type occupying the same length as would type char.

Conclusion. Much more information can be got from reading print-outs of master disc programs of _____.pas.
Note that in an IF-THEN-ELSE statement, no semi-colon must be used in front of Else.
Is there anyone with original Mdex Pascal documentation prepared to make it available for copying?
Finally, Brinch Hansen mentions several console commands. I can't find any which work. Can you?

The Program Copy.

Call from console,
type 2/copy infile,outfile

Copy transfers data from in to out. The files can be disc files with Drive number, or device files, Cons.dev, Print.dev, etc. If you can't remember the correct call, just type Copy . A prompt message is printed.

17·5

```
#p60
111.
112.          {      End Prefix      }
113. (&)
114.      { Global Declarations for Copy.}
115.
116. Const eot = '(:04:)';  nul = '(:0:)';
117.    { These are String Constants but control chars }
118.    { must be written as decimal coded Ascii.}
119. Var s:integer;
120.      ok, console, found: boolean;
121.      attr:fileattr;
122.      text:line;
123.
124.      { Procedure declarations }
125.      { Note: If a procedure calls other procedures, its }
126.      { declaration must follow any for those it calls. }
127.
128. Procedure Context (text:line);
129.  Var i:integer;  c:char;
130.       (local variables only exist whilst the procedure is }
131.      { being executed. They are not accessible from outside.}
132. Begin
133.    i := 0;
134.    Repeat
135.      i := i+1;  c := text[i];
136.      Display (c);
137.    Until c = nul;
138. End;    { of procedure Context }
139.
140. Procedure Help;
141. Begin
142.    Context ('Bad call. Correct form is ''Copy (Infile,Outfile:identifier)''(:10:)');
143. End;
144.
145. Procedure Error (text:line);
146.       (the string passed to this procedure is passed on again to Context)
147. Begin
148.    Context (text);
149.    ok := false;
150. End;
151.
152. Procedure Set_files;  { loop to locate & open both files }
153. Var filename:argtype; i:integer;
154.      {Note this var i is not the same as that used in Context.}
155. Begin
156.    ok := true;
157.    for i := 2 to 3 do
158.     Begin
159.      with param[i] do
160.      if (tag <> idtype) then Help else
161.      begin
162.         Lookup (id, attr, found);
163.         if found then
164.       Begin
165.         Case attr.kind of
166.            scratch,concode,seqcode:
167.              Error ('File kind must be ascii(:10:)');
168.            Ascii:
169.         End;    { of Case }
170. (&)
*
```

```
#P*
170. (&)
171.              If i = 2 then Writearg(inp, param[2]) else
172.                              Writearg(out, param[3]);
173.
174.       End      (of If found...,NB no semi-colon here. )
175.      Else
176.         If i = 2 then Error('IP file unknown(:10:)')
177.                  else Error('OP file unknown(:10:)');
178.      End;    (of If tag...else)
179.    End;      (of loop)
180. End;         (of procedure Set_files)
181.
182. Procedure Terminate;  (Closes files & vets performance.)
183. Begin
184.      Write (eot); write (em);
185.      Readarg (inp,param[2]);  ( close IP  & return status.)
186.      if not param[2].bool then
187.         error('Problem reading IP file(:10:)');
188.      readarg (out, param[3]);  (return OP status )
189.      if not param[3].bool then
190.         error('OP file lost(:10:)');
191.      With param[1] do
192.      Begin tag := booltype; bool := ok end;
193.      ( use the extra param to set performance flag )
194. End; ( of Terminate )
195.
196. Procedure Copytext;
197.      ( Copies standard pages of ascii text.)
198.      ( Declare local variables for use )
199.      ( within this routine only. )
200. Var block:page;  eof:boolean;
201. Begin
202.   Repeat
203.      Readpage (block, eof);
204.      Writepage (block, eof);
205.   Until eof;
206. End;    ( of Copytext )
207.
208.      ( end of all preliminary definitions & declarations )
209.
210.      ( The main program )
211. Begin
212.   Identify ('  Copy:(:10:)'); (pass pgm name to system.)
213.   Set_files;    ( call this procedure )
214.   copytext;     ( call another )
215.   Terminate;    ( and the last.)
216. End.           ( note the mandatory point here )
217.
218.      ( End of Copy )
*Eof*
*
```

SEQUENTIAL PASCAL.

O.C.Walden

Documentary support for Mdex Pascal is very sparse, or horrendously expensive. Brinch Hansen's book [ref.1.], cost me almost $60, but it is not very helpful. These notes are intended to help in getting started. They should be read alongside the User Manual and a print-out of the Prefix from the master disc. The program Copy included works(!) but was written more to illustrate points raised than to have any enduring usefulness.

The Pascal Program.

This must take the form:-
Prefix, followed by Definitions & Declarations of
Constants, Data Types, Variables and Routines, (Procedures & Functions).
These are followed by the sequence of Statements forming the main program which will be executed one at a time.

The program is automatically stored on disc after compilation, and is activated by typing its name and any parameters on the console. To be pedantic, it is the name of the file containing the program which is typed. It is permitted, but surely confusing, to use another for the program.

The program Copy would be called:-

```
Copy (This_file,That_file)
```

File names must comply with Pascal identifier syntax, but may be standard Mdex names. The drive number on disc files is not taken as part of the Pascal identifier. Device files are named as per Mdex, i.e. Cons.dev, Print.dev, etc. This causes some restrictions, see later.

Each parameter in the list, more correctly called the Argument list, may be referenced by number. The system always inserts an extra param of type boolean which may be used to vet program performance. This extra one is Param[1] (or Param(.1.)): so This_file is Param[2], and so on. Ten arguments are allowed, any not mentioned are set to Niltype. See the record type Argtype in Prefix.

The Prefix.

Used by the authors of the language to hive off all hardware related facilities to ease transportability between different implementations. It consists of dummy routines which define the syntax of calls upon them, and direct execution to the actual routines within the operating system. Some routines must be called before others become effective.

File manipulation.

There are two distinct groups for this, they serve to tell the system the names and intended use of the files.

1. READ/WRITE.
   A call on the Prefix procedure Writearg activates this group:-
   Writearg(inp,source) or Writearg(out, dest) opens
   the files named in the variables Source and Dest
   which must be defined as Argtype.
   This activates the procedures Read, Write, Readpage
   and Writepage which may then be used to access
   their files sequentially from top to eof.
   The closure call is Readarg(inp, source) or Readarg(out, dest).
   They return a boolean to inform on satisfactory
   file usage.
2. Open(fileno,filename,found) associates an integer
   with the file by which it is thereafter referenced.
   This can only take the values 1 or 2, so only
   two files can be open at a time.
   This enables Get, Put and Length. Get & Put can
   set the file pointer to any page so give random access.
   Close terminates access and allows re-allocation of the file number.

Console I/O.

If the console is described to the system as Cons.dev, all the above file routines are applicable. The random nature of Get/Put is ignored. However, the console may be addressed by calls to two special routines, Display and Accept. These transfer single Ascii chars and need no prior activation. (see Procedure Context in Copy).

I/O Ascii restriction.

Pascal can only accept ascii chars so all other types will need conversion. Pasedit on the master disc provides much of what is needed for the console. If its small demo program is deleted, it can be copied to a user program in place of Prefix since that is already present. User's global Constants, vars,etc., should be placed between the Prefix and Pasedit. Predefined conversion functions are:-

ORD (x) The ordinal value of the char (x). (its ascii code)
CHR (x) The char whose code is (x).
CONV (x) The Real corresponding to integer (x).
Trunc (x) The Integer corresponding to the real (x).

The Lookup procedure locates a nominated file in the directory and if found returns a boolean True, and the Attributes of the file (see Prefix type Fileattr & Filekind).

Identify tells the system the program name. This is printed on the console before anything else. It is useful as a de-bugging aid, or to indicate the source of results when other programs are called from one running. It is printed once only for each phase. It is optional and is the only use of the pgm name during execution.

Run calls a program from one running. The name and params are exactly as would be typed for a console call, but two extra variables must be supplied as params. They are used after return to the caller, to show Where and How the called pgm was teminated.

Xpeek/Xpoke similar to Basic Peek/Poke.

Xcall calls an Assembler program into action from a running Pascal pgm. (I have not used this yet but it seems simple!

All routines starting IO....

A problem here as most use Prefix descriptions of peripherals such as Typedevice, Printdevice, etc. I have never had these accepted by the compiler which only recognises Mdex Cons.dev, Print.dev.etc. Such routines thus do not work. Exceptions are the types Ioresult and Ioparam. Both of these are effective.

Taskkind. Three tasks are defined, Inputtask, Jobtask & Outputtask. I have never been able to get out of Jobtask. Brinch Hansen says that Readline/Writeline are not effective in Jobtask. They don't work.

Program p(var param:arglist)

This is called when the pgm is activated from the console. No other definition Program must appear.

Other Features. Pre-defined operators for:-

Sets	OR	Union	=	Equal
	&	Intersection	<>	Not equal
	-	Difference	<=	Contained in
	IN	Membership	>=	Contains
	:=	Assignment		

Enumeration types
:= < = > <= <> >=
These have the expected meaning. Results are always Boolean.

Boolean & OR NOT where results are either False or True.

Integer + - * DIV MOD

Real = < > <> <= >= + - * / the pre-defined function ABS(x).

Arrays := = <>

Strings < > <= >=

Records := = <>

Storage requirements in bytes
Enums:2; Reals:8; Sets:16; String of m chars:m;.

Control Chars. These must be written in ascii coded form. e.g. '(:10:)' is the form for Line Feed. They are string vars.

Some Standard Functions. For enumerations,
SUCC (x) and PREC (x) return the successor or predecessor value of x (if there is one).

Arithmetic Functions.
Brinch Hansen Ref[1] is silent on this topic. Some which are implemented are:-
ABS(x), SQR(x) square of x, SIN(X).
these return a result of the same type as 'x', but that must be Real or Integer.
COS(x), ARCTAN(x), LN(x) the natural log.
These must all be Reals.
There must be others, perhaps somebody knows?

Variable & Universal Parameters.
When a procedure is given parameters, it may normally use the value supplied but is not allowed to change it. These are Constant Params. If the procedure is required to change the value, the param identifier must be preceeded by the symbol VAR. e.g. Procedure Something (addr:integer; var block:page)
Of the two params the routine may change the value of 'block', but not 'addr'. Similarly, it cannot always be known in advance what type of data a variable will be called upon to accept. This may be accomodated by writing UNIV in front of the identifier.
e.g. Procedure Something (univ text:line).
This makes 'text' a universal param which can accept any data type occupying the same length as would type char.

Conclusion. Much more information can be got from reading print-outs of master disc programs of _____.pas.
Note that in an IF-THEN-ELSE statement, no semi-colon must be used in front of Else.
Is there anyone with original Mdex Pascal documentation prepared to make it available for copying?
Finally, Brinch Hansen mentions several console commands. I can't find any which work. Can you?

The Program Copy.

Call from console,
type 2/copy infile,outfile

Copy transfers data from in to out. The files can be disc files with Drive number, or device files, Cons.dev, Print.dev, etc. If you can't remember the correct call, just type Copy . A prompt message is printed.

17·5

```
#p60
111.
112.          {     End Prefix ;-)}
113. (&)
114.      { Global Declarations for Copy.}
115.
116. Const eot = '(:04:)'; nul = '(:0:)';
117.    { These are String Constants but control chars }
118.    { must be written as decimal coded Ascii.}
119. Var s:integer;
120.      ok, console, found: boolean;
121.      attr:fileattr;
122.      text:line;
123.
124.      { Procedure declarations }
125.      { Note: If a procedure calls other procedures, its }
126.      { declaration must follow any for those it calls. }
127.
128. Procedure Context (text:line);
129.  Var i:integer; c:char;
130.       {local variables only exist whilst the procedure is }
131.      { being executed. They are not accessible from outside.}
132. Begin
133.    i := 0;
134.    Repeat
135.      i := i+1; c := text[i];
136.      Display (c);
137.    Until c=nl;
138. End;   { of procedure Context }
139.
140. Procedure Help;
141. Begin
142.   Context ('Bad call. Correct form is ''Copy (Infile,Outfile:identifier)''(:10:)');
143. End;
144.
145. Procedure Error (text:line);
146.       {the string passed to this procedure is passed on again to Context}
147. Begin
148.   Context (text);
149.   ok := false;
150. End;
151.
152. Procedure Set_files;  { loop to locate & open both files }
153. Var filename:argtype; i:integer;
154.      {Note this var i is not the same as that used in Context.}
155. Begin
156.   ok := true;
157.   for i := 2 to 3 do
158.    Begin
159.     with param[i] do
160.     if (tag <> idtype) then Help else
161.     begin
162.        Lookup (id, attr, found);
163.        if found then
164.      Begin
165.        Case attr.kind of
166.           scratch,concode,seqcode:
167.            Error ('File kind must be ascii(:10:)');
168.           Ascii;
169.        End;   { of Case }
170. (&)
*
```

```
#P*
170. (&)
171.              If i = 2 then Writearg(inp, param[2]) else
172.                           Writearg(out, param[3]);
173.
174.       End;   (of If found...;NB no semi-colon here. )
175.       Else
176.          If i = 2 then Error('IP file unknown(:10:)')
177.                   else Error('OP file unknown(:10:)');
178.       End;   (of If tag...else)
179.     End;     (of loop)
180. End;         (of procedure Set_files)
181.
182. Procedure Terminate;  (Closes files & vets performance.)
183. Begin
184.     Write (eot); write (em);
185.     Readarg (inp,param[2]);  ( close IP  & return status.)
186.     if not param[2].bool then
187.        error('Problem reading IP file(:10:)');
188.     readarg (out, param[3]);  (return OP status )
189.     if not param[3].bool then
190.        error('OP file lost(:10:)');
191.     With param[1] do
192.     Begin tag := booltype; bool := ok end;
193.     ( use the extra param to set performance flag )
194. End;  (of Terminate)
195.
196. Procedure Copytext;
197.      ( Copies standard pages of ascii text.)
198.      ( Declare local variables for use )
199.      ( within this routine only. )
200. Var block:page;  eof:boolean;
201. Begin
202.   Repeat
203.     Readpage (block, eof);
204.     Writepage (block, eof);
205.   Until eof;
206. End;   ( of Copytext )
207.
208.      ( end of all preliminary definitions & declarations )
209.
210.      ( The main program )
211. Begin
212.   Identify ('  Copy:(:10:)'); (pass pgm name to system.)
213.   Set_files;    ( call this procedure )
214.   copytext;     ( call another )
215.   Terminate;    ( and the last.)
216. End.           ( note the mandatory point here )
217.
218.      ( End of Copy )
*Eof*
*
```

Documentary support for Mdex Pascal is very sparse, or horrendously expensive. Brinch Hansen's book [ref.1.], cost me almost $60, but it is not very helpful. These notes are intended to help in getting started. They should be read alongside the User Manual and a print-out of the Prefix from the master disc. The program Copy included works(!) but was written more to illustrate points raised than to have any enduring usefulness.

The Pascal Program.

This must take the form:-
Prefix, followed by Definitions & Declarations of
Constants, Data Types, Variables and Routines, (Procedures & Functions).
These are followed by the sequence of Statements forming the main program which will be executed one at a time.

The program is automatically stored on disc after compilation, and is activated by typing its name and any parameters on the console. To be pedantic, it is the name of the file containing the program which is typed. It is permitted, but surely confusing, to use another for the program.

The program Copy would be called:-

```
Copy (This_file,That_file)
```

File names must comply with Pascal identifier syntax, but may be standard Mdex names. The drive number on disc files is not taken as part of the Pascal identifier. Device files are named as per Mdex, i.e. Cons.dev, Print.dev, etc. This causes some restrictions, see later.

Each parameter in the list, more correctly called the Argument list, may be referenced by number. The system always inserts an extra param of type boolean which may be used to vet program performance. This extra one is Param[1] (or Param(.1.)); so, This_file is Param[2], and so on. Ten arguments are allowed, any not mentioned are set to Niltype. See the record type Argtype in Prefix.

The Prefix.

Used by the authors of the language to hive off all hardware related facilities to ease transportability between different implementations. It consists of dummy routines which define the syntax of calls upon them, and direct execution to the actual routines within the operating system. Some routines must be called before others become effective.

File manipulation.

There are two distinct groups for this, they serve to tell the system the names and intended use of the files.

1. READ/WRITE.
   A call on the Prefix procedure Writearg activates this group:-
   Writearg(inp,source) or Writearg(out, dest) opens
   the files named in the variables Source and Dest
   which must be defined as Argtype.
   This activates the procedures Read, Write, Readpage
   and Writepage which may then be used to access
   their files sequentially from top to eof.
   The closure call is Readarg(inp, source) or Readarg(out, dest).
   They return a boolean to inform on satisfactory
   file usage.
2. Open(fileno,filename,found) associates an integer
   with the file by which it is thereafter referenced.
   This can only take the values 1 or 2, so only
   two files can be open at a time.
   This enables Get, Put and Length. Get & Put can
   set the file pointer to any page so give random access.
   Close terminates access and allows re-allocation of the file number.

Console I/O.

If the console is described to the system as Cons.dev, all the above file routines are applicable. The random nature of Get/Put is ignored. However, the console may be addressed by calls to two special routines, Display and Accept. These transfer single Ascii chars and need no prior activation. (see Procedure Context in Copy).

I/O Ascii restriction.

Pascal can only accept ascii chars so all other types will need conversion. Pasedit on the master disc provides much of what is needed for the console. If its small demo program is deleted, it can be copied to a user program in place of Prefix since that is already present. User's global Constants, vars,etc., should be placed between the Prefix and Pasedit. Predefined conversion functions are:-

```
ORD (x)   The ordinal value of the char (x). (its ascii code)
CHR (x)   The char whose code is (x).
CONV (x)  The Real corresponding to integer (x).
Trunc (x) The Integer corresponding to the real (x).
```

The Lookup procedure locates a nominated file in the directory and if found returns a boolean True, and the Attributes of the file (see Prefix type Fileattr & Filekind).

Identify tells the system the program name. This is printed on the console before anything else. It is useful as a de-bugging aid, or to indicate the source of results when other programs are called from one running. It is printed once only for each phase. It is optional and is the only use of the pgm name during execution.

Run calls a program from one running. The name and params are exactly as would be typed for a console call, but two extra variables must be supplied as params. They are used after return to the caller, to show Where and How the called pgm was teminated.

Xpeek/Xpoke similar to Basic Peek/Poke.

Xcall calls an Assembler program into action from a running Pascal pgm. (I have not used this yet but it seems simple!

All routines starting IO....

A problem here as most use Prefix descriptions of peripherals such as Typedevice, Printdevice, etc. I have never had these accepted by the compiler which only recognises Mdex Cons.dev, Print.dev.etc. Such routines thus do not work. Exceptions are the types Ioresult and Ioparam. Both of these are effective.

Taskkind. Three tasks are defined, Inputtask, Jobtask & Outputtask. I have never been able to get out of Jobtask. Brinch Hansen says that Readline/Writeline are not effective in Jobtask. They don't work.

Program p(var param:arglist)

This is called when the pgm is activated from the console. No other definition Program must appear.

Other Features. Pre-defined operators for:-

```
Sets    OR      Union          =     Equal
        &     Intersection     <>  Not equal
        -     Difference       <=  Contained in
        IN    Membership       >=  Contains
        :=    Assignment
```

Enumeration types
:= < = > <= <> >=
These have the expected meaning. Results are always Boolean.

Boolean & OR NOT where results are either False or True.

Integer + - * DIV MOD

Real = < > <> <= >= + - * / the pre-defined function ABS(x).

Arrays := = <>

Strings < > <= >=

Records := = <>

Storage requirements in bytes
Enums:2; Reals:8; Sets:16; String of m chars:m;.

Control Chars. These must be written in ascii coded form. e.g. '(:10:)' is the form for Line Feed. They are string vars.

Some Standard Functions. For enumerations,
SUCC (x) and PREC (x) return the successor or predecessor value of x (if there is one).

Arithmetic Functions.
Brinch Hansen Ref[1] is silent on this topic. Some which are implemented are:-
ABS(x), SQR(x) square of x, SIN(X).
these return a result of the same type as x , but that must be Real or Integer.
COS(x), ARCTAN(x), LN(x) the natural log.
These must all be Reals.
There must be others, perhaps somebody knows?

Variable & Universal Parameters.
When a procedure is given parameters, it may normally use the value supplied but is not allowed to change it. These are Constant Params. If the procedure is required to change the value, the param identifier must be preceeded by the symbol VAR. e.g. Procedure Something (addr:integer; var block:page)
Of the two params the routine may change the value of 'block', but not 'addr'. Similarly, it cannot always be known in advance what type of data a variable will be called upon to accept. This may be accomodated by writing UNIV in front of the identifier.
e.g. Procedure Something (univ text:line).
This makes 'text' a universal param which can accept any data type occupying the same length as would type char.

Conclusion. Much more information can be got from reading print-outs of master disc programs of _____.pas.
Note that in an IF-THEN-ELSE statement, no semi-colon must be used in front of Else.
Is there anyone with original Mdex Pascal documentation prepared to make it available for copying?
Finally, Brinch Hansen mentions several console commands. I can't find any which work. Can you?

The Program Copy.

Call from console,
type 2/copy infile,outfile

Copy transfers data from in to out. The files can be disc files with Drive number, or device files, Cons.dev, Print.dev, etc. If you can't remember the correct call, just type Copy . A prompt message is printed.

17·5

```
#p60
111.
112.          {      End Prefix      }
113. (&)
114.      { Global Declarations for Copy.}
115.
116. Const eot = '(:04:)';  nul = '(:0:)';
117.    { These are String Constants but control chars }
118.    { must be written as decimal coded Ascii.}
119. Var s:integer;
120.      ok, console, found: boolean;
121.      attr:fileattr;
122.      text:line;
123.
124.      { Procedure declarations }
125.      { Note: If a procedure calls other procedures, its }
126.      { declaration must follow any for those it calls. }
127.
128. Procedure Context (text:line);
129.  Var i:integer;  c:char;
130.       (local variables only exist whilst the procedure is }
131.      { being executed. They are not accessible from outside.}
132. Begin
133.    i := 0;
134.    Repeat
135.      i := i+1;  c := text[i];
136.      Display (c);
137.    Until c = nl;
138. End;    { of procedure Context }
139.
140. Procedure Help;
141. Begin
142.    Context ('Bad call. Correct form is ''Copy (Infile,Outfile:identifier)''(:10:)');
143. End;
144.
145. Procedure Error (text:line);
146.       (the string passed to this procedure is passed on again to Context}
147. Begin
148.    Context (text);
149.    ok := false;
150. End;
151.
152. Procedure Set_files;  { loop to locate & open both files }
153. Var filename:argtype; i:integer;
154.      {Note this var i is not the same as that used in Context.}
155. Begin
156.    ok := true;
157.    for i := 2 to 3 do
158.     Begin
159.      with param[i] do
160.      if (tag <> idtype) then Help else
161.      begin
162.         Lookup (id, attr, found);
163.         if found then
164.       Begin
165.         Case attr.kind of
166.            scratch,concode,seqcode:
167.             Error ('File kind must be ascii(:10:)');
168.            Ascii:
169.         End;    { of Case }
170. (&)
*
```

```
#P*
170. (&)
171.              If i = 2 then Writearg(inp, param[2]) else
172.                            Writearg(out, param[3]);
173.
174.       End      (of If found...,NB no semi-colon here. )
175.      Else
176.         If i = 2 then Error('IP file unknown(:10:)')
177.                  else Error('OP file unknown(:10:)');
178.      End;    (of If tag...else)
179.     End;     (of loop)
180. End;         (of procedure Set_files)
181.
182. Procedure Terminate;  (Closes files & vets performance.)
183. Begin
184.      Write (eot); write (em);
185.      Readarg (inp,param[2]);  ( close IP  & return status.)
186.      if not param[2].bool then
187.         error('Problem reading IP file(:10:)');
188.      readarg (out, param[3]);  (return OP status )
189.      if not param[3].bool then
190.         error('OP file lost(:10:)');
191.      With param[1] do
192.      Begin tag := booltype; bool := ok end;
193.      ( use the extra param to set performance flag )
194. End; ( of Terminate )
195.
196. Procedure Copytext;
197.      ( Copies standard pages of ascii text.)
198.      ( Declare local variables for use )
199.      ( within this routine only. )
200. Var block:page;  eof:boolean;
201. Begin
202.   Repeat
203.     Readpage (block, eof);
204.     Writepage (block, eof);
205.   Until eof;
206. End;    ( of Copytext )
207.
208.      ( end of all preliminary definitions & declarations )
209.
210.      ( The main program )
211. Begin
212.   Identify ('  Copy:(:10:)'); (pass pgm name to system.)
213.   Set_files;    ( call this procedure )
214.   copytext;     ( call another )
215.   Terminate;    ( and the last.)
216. End.           ( note the mandatory point here )
217.
218.      ( End of Copy )
*Eof*
*
```

Documentary support for Mdex Pascal is very sparse, or horrendously expensive. Brinch Hansen's book [ref.1.], cost me almost $60, but it is not very helpful. These notes are intended to help in getting started. They should be read alongside the User Manual and a print-out of the Prefix from the master disc. The program Copy included works(!) but was written more to illustrate points raised than to have any enduring usefulness.

The Pascal Program.

This must take the form:-
Prefix, followed by Definitions & Declarations of
Constants, Data Types, Variables and Routines, (Procedures & Functions).
These are followed by the sequence of Statements forming the main program which will be executed one at a time.

The program is automatically stored on disc after compilation, and is activated by typing its name and any parameters on the console. To be pedantic,it is the name of the file containing the program which is typed. It is permitted, but surely confusing, to use another for the program.

The program Copy would be called:-

```
Copy (This_file,That_file)
```

File names must comply with Pascal identifier syntax, but may be standard Mdex names. The drive number on disc files is not taken as part of the Pascal identifier. Device files are named as per Mdex, i.e. Cons.dev, Print.dev, etc. This causes some restrictions, see later.

Each parameter in the list, more correctly called the Argument list, may be referenced by number. The system always inserts an extra param of type boolean which may be used to vet program performance. This extra one is Param[1] (or Param(.1.)): so This_file is Param[2], and so on. Ten arguments are allowed, any not mentioned are set to Niltype. See the record type Argtype in Prefix.

The Prefix.

Used by the authors of the language to hive off all hardware related facilities to ease transportability between different implementations. It consists of dummy routines which define the syntax of calls upon them, and direct execution to the actual routines within the operating system. Some routines must be called before others become effective.

File manipulation.

There are two distinct groups for this, they serve to tell the system the names and intended use of the files.

1. READ/WRITE.
   A call on the Prefix procedure Writearg activates this group:-
   Writearg(inp,source) or Writearg(out, dest) opens
   the files named in the variables Source and Dest
   which must be defined as Argtype.
   This activates the procedures Read, Write, Readpage
   and Writepage which may then be used to access
   their files sequentially from top to eof.
   The closure call is Readarg(inp, source) or Readarg(out, dest).
   They return a boolean to inform on satisfactory
   file usage.
2. Open(fileno,filename,found) associates an integer
   with the file by which it is thereafter referenced.
   This can only take the values 1 or 2, so only
   two files can be open at a time.
   This enables Get, Put and Length. Get & Put can
   set the file pointer to any page so give random access.
   Close terminates access and allows re-allocation of the file number.

Console I/O.

If the console is described to the system as Cons.dev, all the above file routines are applicable. The random nature of Get/Put is ignored. However, the console may be addressed by calls to two special routines, Display and Accept. These transfer single Ascii chars and need no prior activation. (see Procedure Context in Copy).

I/O Ascii restriction.

Pascal can only accept ascii chars so all other types will need conversion. Pasedit on the master disc provides much of what is needed for the console. If its small demo program is deleted, it can be copied to a user program in place of Prefix since that is alrady present. User's global Constants, vars,etc., should be placed between the Prefix and Pasedit. Predefined conversion functions are:-

```
ORD (x)   The ordinal value of the char (x). (its ascii code)
CHR (x)   The char whose code is (x).
CONV (x)  The Real corresponding to integer (x).
Trunc (x) The Integer corresponding to the real (x).
```

The Lookup procedure locates a nominated file in the directory and if found returns a boolean True, and the Attributes of the file (see Prefix type Fileattr & Filekind).

Identify tells the system the program name. This is printed on the console before anything else. It is useful as a de-bugging aid, or to indicate the source of results when other programs are called from one running. It is printed once only for each phase. It is optional and is the only use of the pgm name during execution.

Run calls a program from one running. The name and params are exactly as would be typed for a console call, but two extra variables must be supplied as params. They are used after return to the caller, to show Where and How the called pgm was teminated.

Xpeek/Xpoke similar to Basic Peek/Poke.

Xcall calls an Assembler program into action from a running Pascal pgm. (I have not used this yet but it seems simple!

All routines starting IO....

A problem here as most use Prefix descriptions of peripherals such as Typedevice, Printdevice, etc. I have never had these accepted by the compiler which only recognises Mdex Cons.dev, Print.dev.etc. Such routines thus do not work. Exceptions are the types Ioresult and Ioparam. Both of these are effective.

Taskkind. Three tasks are defined, Inputtask, Jobtask & Outputtask. I have never been able to get out of Jobtask. Brinch Hansen says that Readline/Writeline are not effective in Jobtask. They don't work.

Program p(var param:arglist)

This is called when the pgm is activated from the console. No other definition Program must appear.

Other Features. Pre-defined operators for:-

```
Sets    OR      Union          =     Equal
        &     Intersection     <>  Not equal
        -     Difference       <=  Contained in
        IN    Membership       >=  Contains
        :=    Assignment
```

Enumeration types
:= < = > <= <> >=
These have the expected meaning. Results are always Boolean.

Boolean & OR NOT where results are either False or True.

Integer + - * DIV MOD

Real = < > <> <= >= + - * / the pre-defined function ABS(x).

Arrays := = <>

Strings < > <= >=

Records := = <>

Storage requirements in bytes
Enums:2; Reals:8; Sets:16; String of m chars:m;.

Control Chars. These must be written in ascii coded form. e.g. '(:10:)' is the form for Line Feed. They are string vars.

Some Standard Functions. For enumerations,
SUCC (x) and PREC (x) return the successor or predecessor value of x (if there is one).

Arithmetic Functions.
Brinch Hansen Ref[1] is silent on this topic. Some which are implemented are:-
ABS(x), SQR(x) square of x, SIN(X).
these return a result of the same type as x , but that must be Real or Integer.
COS(x), ARCTAN(x), LN(x) the natural log.
These must all be Reals.
There must be others, perhaps somebody knows?

Variable & Universal Parameters.
When a procedure is given parameters, it may normally use the value supplied but is not allowed to change it. These are Constant Params. If the procedure is required to change the value, the param identifier must be preceeded by the symbol VAR. e.g. Procedure Something (addr:integer; var block:page)
Of the two params the routine may change the value of 'block', but not 'addr'. Similarly, it cannot always be known in advance what type of data a variable will be called upon to accept. This may be accomodated by writing UNIV in front of the identifier.
e.g. Procedure Something (univ text:line).
This makes 'text' a universal param which can accept any data type occupying the same length as would type char.

Conclusion. Much more information can be got from reading print-outs of master disc programs of _____.pas.
Note that in an IF-THEN-ELSE statement, no semi-colon must be used in front of Else.
Is there anyone with original Mdex Pascal documentation prepared to make it available for copying?
Finally, Brinch Hansen mentions several console commands. I can't find any which work. Can you?

The Program Copy.

Call from console,
type 2/copy infile,outfile

Copy transfers data from in to out. The files can be disc files with Drive number, or device files, Cons.dev, Print.dev, etc. If you can't remember the correct call, just type Copy . A prompt message is printed.

17·5

```
#p60
111.
112.          {     End Prefix     }
113. (&)
114.      { Global Declarations for Copy.}
115.
116. Const eot = '(:04:)';  nul = '(:0:)';
117.    { These are String Constants but control chars }
118.    { must be written as decimal coded Ascii.}
119. Var s:integer;
120.      ok, console, found: boolean;
121.      attr:fileattr;
122.      text:line;
123.
124.      { Procedure declarations }
125.      { Note: If a procedure calls other procedures, its }
126.      { declaration must follow any for those it calls. }
127.
128. Procedure Context (text:line);
129.   Var i:integer;  c:char;
130.       {local variables only exist whilst the procedure is }
131.      { being executed. They are not accessible from outside.}
132. Begin
133.    i := 0;
134.    Repeat
135.      i := i+1;  c := text[i];
136.      Display (c);
137.    Until c = nul
138. End;    { of procedure Context }
139.
140. Procedure Help;
141. Begin
142.    Context ('Bad call. Correct form is ''Copy (Infile,Outfile:identifier)''(:10:)');
143. End;
144.
145. Procedure Error (text:line);
146.       {the string passed to this procedure is passed on again to Context}
147. Begin
148.    Context (text);
149.    ok := false;
150. End;
151.
152. Procedure Set_files;  { loop to locate & open both files }
153. Var filename:argtype; i:integer;
154.      {Note this var i is not the same as that used in Context.}
155. Begin
156.    ok := true;
157.    for i := 2 to 3 do
158.     Begin
159.      with param[i] do
160.      if (tag <> idtype) then Help else
161.      begin
162.         Lookup (id, attr, found);
163.         if found then
164.       Begin
165.         Case attr.kind of
166.            scratch,concode,seqcode:
167.             Error ('File kind must be ascii(:10:)');
168.            Ascii:
169.         End;    { of Case }
170. (&)
*
```

```
#P*
170. (&)
171.              If i = 2 then Writearg(inp, param[2]) else
172.                            Writearg(out, param[3]);
173.
174.       End     {of If found...,NB no semi-colon here. }
175.      Else
176.          If i = 2 then Error('IP file unknown(:10:)')
177.                   else Error('OP file unknown(:10:)');
178.      End;    {of If tag...else}
179.    End;      {of loop}
180. End;         {of procedure Set_files}
181.
182. Procedure Terminate;  {Closes files & vets performance.}
183. Begin
184.      Write (eot); write (em);
185.      Readarg (inp,param[2]);  { close IP  & return status.}
186.      if not param[2].bool then
187.         error('Problem reading IP file(:10:)');
188.      readarg (out, param[3]);  {return OP status }
189.      if not param[3].bool then
190.         error('OP file lost(:10:)');
191.      With param[1] do
192.      Begin tag := booltype; bool := ok end;
193.      { use the extra param to set performance flag }
194. End;  {of Terminate }
195.
196. Procedure Copytext;
197.      { Copies standard pages of ascii text.}
198.      { Declare local variables for use }
199.      { within this routine only. }
200. Var block:page;  eof:boolean;
201. Begin
202.   Repeat
203.      Readpage (block, eof);
204.      Writepage (block, eof);
205.   Until eof;
206. End;    { of Copytext }
207.
208.      { end of all preliminary definitions & declarations }
209.
210.      { The main program }
211. Begin
212.   Identify ('  Copy:(:10:)'); {pass pgm name to system.}
213.   Set_files;    { call this procedure }
214.   copytext;     { call another }
215.   Terminate;    { and the last.}
216. End.           { note the mandatory point here }
217.
218.      { End of Copy }
*Eof*
*
```