Pascal/MT+

Release 5

Language Reference
and
Applications Guide

MT MicroSYSTEMS
1562 Kings Cross Drive
Cardiff-by-the-sea, CA. 92007

COPYRIGHT

DISCLAIMER

MT MicroSYSTEMS makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, MT MicroSYSTEMS reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of MT MicroSYSTEMS to notify any person of such revision or changes.

TRADEMARKS

Pascal/MT+ is a trademark of MT MicroSYSTEMS. Credit is given to Digital Research of California for its trademarks: CP/M, MP/M-80, SID, and MAC. Credit is given to Microsoft of Washington state for its trademarks: MACRO-80, FORTRAN-80 and LINK-80. Any reference to CP/M, MP/M, SID, MAC, M80, FORTRAN, and L80 also refer to the appropriate trademarked software packages.

| Second printing: October 1980 |

Table of Contents

Table of Contents

Pascal/MT+ APPLICATIONS GUIDE:

Table of Contents

0.0 System Overview

PLEASE NOTE: THE PURPOSE OF THIS DOCUMENTATION IS NOT TO TEACH THE Pascal LANGUAGE BUT RATHER TO DESCRIBE IN DETAIL THE SPECIFIC IMPLEMENTATION CALLED Pascal/MT+. WE STRONGLY RECOMMEND THAT THE USER PURCHASE EITHER Jensen and Wirth or Addyman and Wilson AS Pascal TEXTS (BOTH ARE AVAILABLE FROM BOOKSTORES AND BYTE MAGAZINE, AND BOTH ARE PUBLISHED BY SPRINGER / VERLAG IN NEW YORK CITY.

Contained in this manual is the documentation for the Pascal/MT+ software system. The Pascal/MT+ package consists of the following software components:

Pascal/MT+	compiler
Link/MT+	linker
PASLIB.ERL	run-time library relocatable object file
RTP/MT+	run-time library source file
Debug/MT+	run-time debugging tool
Disasm/MT+	disassembler
Patch/MT+	System patch application program

Also included is a group of utility programs written in Pascal/MT+ which are included for user information as well as their intrinsic value as tools.

The Pascal/MT+ system has evolved from an original goal as an assembly language replacement tool into a full ISO (International Standards Organization) Standard Pascal system including capability for modular compilation. The Pascal/MT+ compiler is a completely new compiler designed from the beginning to implement the entire Pascal language and is not a revision of our popular Pascal/MT package. Pascal/MT+ and Pascal/MT have been used for such diverse applications as multi-processor measurment machines, process controllers, business applications and software tool development (compilers, assemblers, etc.). All of the features and facilities present in our Pascal/MT system which has been sold to over 1000 users are present in Pascal/MT+. Conformance to the ISO standard has required some syntactic changes from Pascal/MT but the functionality of our extensions to the Pascal language remain the same as Pascal/MT.

The Pascal/MT+ system is designed to be an effective tool both in computer resource and human resource utilization. The Pascal/MT+ compiler, linker, debugger and disassembler have been designed with an eye to practical user needs such as minimum waiting time and visibility. We are dedicated to the construction of useful software "power tools" which amplify the creative power of the programmer/engineer. If after having read this manual and/or having used the Pascal/MT+ system, you have any ideas for improving the usability of our software please write or call. We would be glad to hear from you.

Pascal/MT+ Language Guide

1.Ø Introduction

The purpose of this language guide is to define the language features of Pascal/MT+. This guide assumes that the reader is familiar with the Jensen and Wirth and/or the ISO draft standard (currently DPS/7185). The standard Pascal features which are different in Pascal/MT+ than those in the standard and in Jensen and Wirth's 'Report' are described by section. In each section BNF (Backus Normal Form) syntax is provided for reference. The complete BNF description of the language is present in section 16.3 of the applications guide. Each section corresponds to Wirth's 'Report' beginning with section 2.

2.0 Summary of the language

The language compiled by Pascal/MT+ is identical to the ISO draft standard (DPS/7185 as of 10/1/80) with the following additions:

Additional pre-defined scalar types: BYTE, WORD, STRING
Expressions may contain the pre-declared INP array
Assignments may be made to the pre-declared OUT array
For 16-bit CPU systems INPW and OUTW for WORD I/O
Operators on integers & (and), !,| (or), and ~,\,? (not)
Else on CASE statement
Interrupt and External procedures
Additional built-in procedures and functions
Modular compilation facilities
Re-directable I/O facilities (user written char I/O)

3.Ø Notation, terminology, and vocabulary

```
<letter> ::= A | B | C | D | E | F | G | H | I | J |
             K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z | a | b | c | d |
             e | f | g | h | i | j | k | l | m | n |
             n | o | p | q | r | s | t | u | v | w |
             x | y | z | @

<digit>  ::= Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
             A | B | C | D | E | F   {only allowed in HEX numbers}

<special symbol> ::= {reserved words are listed in section 16.2}
             +  |  -  |  *  |  /  |  =  |  <> |  <  |  >  |
             <= |  >= |  (  |  )  |  [  |  ]  |  {  |  }  |
             := |  .  |  ,  |  ;  |  :  |  '  |  ^  |

             {the following are additional or substitutions:}
             (. |  .) |  ~  |  \  |  ?  |  !  |     |  $  |  &

             (. is a synonym for [
             .) is a synonym for ]
             ~, \, and ? are synonyms  (see section 8.1.1)
             !, and | are synonyms     (see section 8.1.2)
             &                         (see section 8.1.3)
```

The symbol '@' is a legal letter in addition to those listed in the 'Report'. This has been added because all of our run-time library routine are written using this special character and this allowed us to decide which routines should be written in Pascal and which should be written in assembly language.

A comment beginning with '(*' must end with '*)'. A comment beginning with '{' must end with '}'. To allow nested comments the begin comment delimiter must be the same as the end comment delimiter. Thus, in Pascal/MT+ the following is legal:

```
(* outer comment ...{ inner comment....}...outer comment *)
```

4.0 Identifiers, numbers, and strings

```
<identifier>        ::= <letter> {<letter or digit or underscore>}
<letter or digit>   ::= <letter> | <digit> | _

<digit sequence>    ::= <digit> {<digit>}
<unsigned integer>  ::= $ <digit sequence> |
                        <digit sequence>
<unsigned real>     ::= <unsigned integer> . <digit sequence>    |
                        <unsigned integer> . <digit sequence>
                        E <scale factor>                         |
                        <unsigned integer> E <scale factor>
<unsigned number>   ::= <unsigned integer | <unsigned real>
<scale factor>      ::= <unsigned integer> | <sign><unsigned integer>
<sign>              ::= + | -

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

All identifiers are significant to 8 characters. External identifiers are significant to either six or seven characters depending upon usage (see section 14 of the language guide and section 13.2 of the applications guide) The underscore character (_) is legal between letters and digits in an identifier and is ignored by the compiler (i.e. A_B is equivalent to AB). Identifiers may begin with an '@'. This is to allow declaration of external run-time routines within a Pascal program. Users are, in general, advised to avoid the '@' character to eliminate the chance of conflict with run-time routine names.

Numbers may be hex as well as decimal. Placing a '$' in front of an integer number causes it to be interpreted as a hex number by the compiler. The symbol <digit> now includes: 'A', 'B', 'C', 'D', 'E' and 'F'. These may be upper or lower case.

6.Ø Data type difinitions

```
<type>            ::= <simple type>     |
                      <structured type> |
                      <pointer type>
<type definition> ::= <identifier> = <type>
```

6.1 Simple types

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

6.1.1 Scalar types

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

6.1.2 Standard types

The following types are standard in Pascal/MT+

```
INTEGER
REAL
BOOLEAN
CHAR

BYTE
WORD
STRING
```

Three additional standard types exist in Pascal/MT+. See the applications guide for information on representation and usage of all standard and structured types.

STRING : Packed array [Ø..n] of char;
byte Ø is dynamic length byte
bytes 1..n are characters.

BYTE : Subrange Ø..255 with special attribute that it is compatible also with CHAR type

WORD : Unsigned native machine word.
Guaranteed to be the same size as a pointer.
(integers and pointers are different sizes in some 16/32 bit machines).

6.1.3 Subrange types

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

6.2 Structured types

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

The reserved word PACKED is detected and handled by the Pascal/MT+ compiler as follows:

> All structures are packed at the BYTE level even if the PACKED reserved word is not found. The user is refered to section 13.Ø in the applications guide for a description of how fields and contiguous variables are allocated for various target machines.

6.2.1 Array types

```
<array type>      ::= <normal array> |
                      <string array>
<string array>    ::= STRING <max length>
<max length>      ::= [ <intconst> ] |
                      <empty>
<intconst>        ::= <unsigned integer> |
                      <int const id>
<int const id>    ::= <identifier>
<normal array>    ::= ARRAY [ <index type> {,<index type>}] OF
                      <component type>
<index type>      ::= <simple type>
<component type>  ::= <type>
```

Variables of type STRING have a default length of 81 bytes (8Ø data characters). A different length can be specified in square brackets following the word STRING. The length must be a constant (either literal or declared e.g. STRING[5] or STRING[xyz] (where xyz is a constant (xyz=1Ø))) and represents the length of the DATA portion (i.e. one more byte is actually allocated for the length).

6.2.2 Record types

```
<record type>      ::= RECORD <field list> END
<field list>       ::= <fixed part>                    |
                       <fixed part> ; <variant part>   |
                       <variant part>
<fixed part>       ::= <record section> {;<record section>}
<record section>   ::= <field identifier> {,<field identifier>} : <type> |
                       <empty>
<variant part>     ::= CASE <tag field> <type identifier> OF
                       <variant> {;<variant>}
<variant>          ::= <case label list> : (<field list>) |
                       <empty>
<case label list>  ::= <case label> {,<case label>}
<case label>       ::= <constant>
<tag field>        ::= <identifier> : |
                       <empty>
```

6.2.3 Set types

```
<set type>  ::= SET OF <base type>
<base type> ::= <simple type>
```

The maximum range of a base type is 0..255. For example, a set of [0..10000] is not legal but the set of CHAR or set of 0..255 is legal but set of 0..256 is not.

6.2.4 File types

<file type> ::= file {of <type>}

Untyped files are allowed. They are used for CHAINING (see 9.Ø in applications guide) and are also used with BLOCKREAD and BLOCKWRITE procedures (see 5.18 in applications guide). The user should be extremely careful when using untyped files.

When wishing to read a file of ASCII characters and using implied conversions for integers and reals the user should use the pre-defined type TEXT. TEXT is NOT exactly the same as FILE OF CHAR but has conversion implied in READ and WRITE procedure calls and also may be used with READLN and WRITELN.

6.3 Pointer types

<pointer type> ::= ^ <type identifier>

Pointer types are identical to the standard except that weak type checking exists when the RELAXED type checking feature of the compiler is enabled (the default) (see sections 2.6 in the applications guide). In this case pointers and WORDs used as pointers are compatible in all cases.

7.0 Declarations and denotations of variables

```
<variable>      ::= <var>                  |
                    <external var>         |
                    <absolute var>

<external var>  ::= EXTERNAL <var>

<absolute var>  ::= ABSOLUTE [ <constant> ] <var>

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

ABSOLUTE variables may be declared if the user knows the address at compile time. The user declares variable(s) to be absolute using special syntax in a VAR declaration. ABSOLUTE variables are not allocated any space in the user's data segment by the compiler and the user is responsible for making sure that no compiler allocated variables conflict with the absolute variables. NOTE: STRING VARIABLES MAY NOT EXIST AT LOCATIONS <= 100H. This is done so that the run-time routines can detect the difference between a string address and a character on the top of the stack. Characters have the high byte of 0 when present on the stack. After the colon (:) and before the type of the variable(s) the user places the keyword ABSOLUTE followed by the address of the variable in brackets ([...]):

Examples:

```
I:      ABSOLUTE [$8000] INTEGER;
SCREEN: ABSOLUTE [$C000] ARRAY[0..15] OF ARRAY[0..63] OF CHAR;
```

7.1 Entire variables

```
<entire variable>     ::= <variable identifier>
<variable identifier> ::= <identifier>
```

7.2. Component variables

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

7.2.1 Indexed variables

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

STRING variables are to be treated as a PACKED array of CHAR for subscripting purposes. The valid range is 0..maxlength where maxlength is 80 for a default length (see section 7.0).

7.2.2 Field designators

```
<field designator> ::= <record variable> . <field identifier>
<record variable>  ::= <variable>
<field identifier> ::= <identifier>
```

7.2.3 File buffers

```
<file buffer>   ::= <file variable> ^
<file variable> ::= <variable>
```

7.3 Referenced variables

```
<referenced variable> ::= <pointer variable> ^
<pointer variable>    ::= <variable>
```

8.0 Expressions

```
<unsigned constant> ::= <unsigned number>
                        <string>
                        NIL
                        <constant identifier>
<factor>            ::= <variable>
                        <unsigned constant>
                        <function designator>
                        ( <expression> )
                        NOT <factor>
<set>               ::= [ <element list> ]
<element list>      ::= <element> {,<element>}
                        <empty>
<element>           ::= <expression>
                        <expression> .. <expression>
<term>              ::= <factor> <multiplying operator> <factor>
<simple expression> ::= <term>
                        <simple expression> <adding operator> <term>
                        <adding operator> <term>
<expression>        ::= <simple expression>
                        <simple expression> <relational operator>
                            <simple expression>
```

The pre-declared array INP is used in expressions to return a byte from an I/O port. The INP array is of type BYTE and therefore may be used with integers and CHAR variables. The array is indexed by an expression. If the expression is a constant the compiler will generate in-line code for the port access. Otherwise a subroutine is called for variable port numbers. The INPW array is present on the 16-bit CPU system for WORD oriented input ports. Allowable range for port numbers is CPU dependent, 0..255 for 8080/Z80, see the specific processor applications guide for more information on non-8080 type CPUs.

Example:

```
x := INP[$55];

x := INP[baseaddr+9];


x may be of type BYTE, CHAR or INTEGER
```

An additional category of operators on 16-bit variables are &,! (also |), and ~ (also \ and ?) denoting AND, OR and ONE's complement NOT, respectively. These have the same precedence as their equivalent boolean operators and accept any type of operand with a size <= 2 bytes.

8.1 Operators

8.1.1 The operator not

<logical not operator> ::= NOT | ~ | \ | ?

~ (synonyms \ and ?) is a NOT operator for non-booleans.

8.1.2 Multiplying operators

<multiplying operator> ::= * | / | DIV | MOD | AND | &

& is an AND operator on non-booleans.

8.1.3 Adding operators

<adding operator> ::= + | - | OR | | | !

! (synonym |) is an OR operator on non-booleans.

8.1.4 Relational operators

<relational operators> ::= = | <> | < | <= | > | >= | IN

8.2 Function designators

<function designator> ::= <function identifier> |
<function identifier> (<parm> {,<parm>)
<function identifier> ::= <identifier>

9.0 Statements

```
<statement>              ::= <label> : <unlabelled statement> |
                             <unlabelled statement>
<unlabelled statement>   ::= <simple statement>               |
                             <structured statement>
<label>                  ::= <unsigned integer>
```

9.1 Simple Statements

```
<simple statement ::= <assignment statement> |
                      <procedure statement>  |
                      <goto statement>       |
                      <empty statement>
<empty statement> ::= <empty>
```

9.1.1 Assignment statements

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

Pascal/MT+ implements a pre-declared BYTE array called OUT to which may be assigned items of type integer, byte, or char. OUT is indexed by an expression. The range is CPU dependent and is 0..255 for 8080, 8085, and Z80. For 16-bit CPU systems the pre-declared WORD array OUTW is also present. For 8085 systems the system accepts the strings: RIM85 and SIM85 as subscripts for INP and OUT. RIM85 and SIM85 are not stored in the symbol table but are examined for using a string comparision therefore users not compiling to an 8085 target machine are not penalized. Consult the CPU applications guide for more information. As in the case of INP, if the expression is a constant the compiler will generate in-line code for the port access. Otherwise a subroutine is called to handle variable port numbers.

```
OUT[portnum] := $88;
```

To the list of exceptions to assignment compatibility add:

1. Integer expressions may be assigned to variables of type pointer. For example:

```
TYPE X = RECORD
           (* field declarations *)
         END;
```

```
VAR P : ^X;
    I : INTEGER;
.................
P := I+1;
```

2. Expressions of type CHAR may be assigned to variables of type STRING.

3. Variables of type CHAR and literal characters may be assigned to variables of type BYTE.

4. Expressions evaluating to the type WORD may be assigned to pointer variables.

5. Expressions evaluating to the type INTEGER may be assigned to variables of type WORD

9.1.2 Procedure statements

```
<procedure statement> ::= <procedure identifier> ( <parm> {,<parm>} ) |
                          <procedure identifier>
<procedure identifier>::= <identifier>
<parm>                ::= <procedure identifier> |
                          <function identifier>  |
                          <expression>           |
                          <variable>
```

9.1.3 Goto statements

```
<goto statement> ::= goto <label>
```

9.2 Structured statements

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

9.2.1 Compound statments

```
<compound statement> ::= BEGIN <statement> {,<statement>} END
```

9.2.2 Conditional statments

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

9.2.2.1 If statements

```
<if statement> ::= IF <expression> THEN <statement> ELSE <statement> |
                   IF <expression> THEN <statement>
```

9.2.2.2 Case statements

```
<case statement> ::= CASE <expression> OF
                     <case list> {,<case list>}
                     {ELSE <statement>}
                     END

<case list>      ::= <label list> : <statement> |
                     <empty>
<label list>     ::= <case label> {,<case label>}
```

Pascal/MT+ implements an ELSE clause on the case statement. In addition if the selecting expression does not match any of the case selectors the program flow will "drop through" the case statement. This is different than the standard which says this condition is an error.

Example:

```
CASE CH OF
  'A' : WRITELN('A');
  'Q' : WRITELN('Q');
  ELSE
    WRITELN('NOT A OR Q')
END
```

9.2.3 Repetitive statements

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

9.2.3.1 While statements

```
<while statement> ::= WHILE <expression> DO <statement>
```

9.2.3.2 Repeat statements

```
<repeat statement> ::= REPEAT <statement> {,<statement>} UNTIL <expression>
```

9.2.3.3 For statements

```
<for statement> ::= FOR <ctrlvar> := <for list> DO <statement>
<for list>      ::= <expression> DOWNTO <expression> |
                    <expression>   TO   <expression>
<ctrlvar>       ::= <variable>
```

9.2.4 With statements

```
<with statement>       ::= WITH <record variable list> DO <statment>
<record variable list> ::= <record variable> {,<record variable>}
```

The user should note that the ISO standard differs from Jensen and Wirth in that only LOCAL variables are allowed as FOR loop control variables. This prevents such programming errors as the inadvertant use of a GLOBAL variable as a FOR control variable when burried 5 levels deep in nesting.

In a recursive stack frame environment the user is limited to 16 FOR and / or WITH statements in a single procedure / function. This is so that the compiler can allocate a fixed number of temporary locations (16 words) in the data segment for the procedure / function. This environment is present in all CPUs except the 8080 / Z80 default environment (static allocation). The 8080 / Z80 enter the stack frame environment using the $S switch.

10.0 Procedure declarations

```
<procedure declaration>      ::= EXTERNAL <procedure heading> |
                                 <procedure heading> <block>
<block>                      ::= <label declaration part>
                                 <constant definition part>
                                 <type definition part>
                                 <variable declaration part>
                                 <procfunc declaration part>
                                 <statement part>

<procedure heading>          ::= PROCEDURE <identifier> <parmlist>     |
                                 PROCEDURE <identifier> ;              |
                                 PROCEDURE INTERRUPT [ <constant> ] ;

<parmlist>                   ::= ( <fparm> {,<fparm>} )

<fparm>                      ::= <procedure heading>     |
                                 <function heading>     |
                                 VAR <parm group>       |
                                 <parm group>

<parm group>                 ::= <identifier> {,<identifier>} :
                                   <type identifier>             |
                                 <identifier> {,<identifier>} :
                                   <conformant array>

<conformant array>           ::= ARRAY [ <indxtyp> {;<indxtyp} ] OF
                                 <conarray2>

<conarray2>                  ::= <type identifier>   |
                                 <conformant array>

<indxtyp>                    ::= <identifier> .. <identifier> : <ordtypid>

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

<scalar type identifier>     ::= <identifier>

<subrange type identifier>   ::= <identifier>

<label declaration part>     ::= <empty> |
                                 LABEL <label> {,<label>} ;

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

<type definition part>       ::= <empty> |
                                 TYPE
                                   <type definition>
```

```
                                    {;<type definition>} ;

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

<procfunc part>             ::= {<proc or func> ; }

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

<statement part>            ::= <compound statement>
```

A special procedure type is implemented in Pascal/MT+, the interrupt procedure. The user selects the vector to be associated with each interrupt. The procedure is declared as follows:

```
PROCEDURE INTERRUPT[vector number] procname;
```

The user is referred to section 6.Ø of the applications guide for more information on using INTERRUPT procedures.

The user should note that the ISO standard has added the CONFORMANT ARRAY SCHEMA for passing arrays of similar structure (i.e. same number of dimensions, compatible index type, ans same element type), but different upper and lower bounds. The user may now pass, for example, an array dimensioned as 1..1Ø and an array 2..5Ø to a procedure which expecting an array. The user defines the array as a VAR parameter and in the process of declaring the array the user defines also variables to hold the upper and lower bound of the array. These upper and lower bound items are filled in at RUN-TIME by the generated code. The user should note that in order to pass arrays in this manner the index type must be compatible with the type of the conformant array bounds.

Below is an example of passing two arrays to a procedure which displays the contents of the arrays on the file OUTPUT:

```
PROGRAM DEMOCON;

TYPE
  NATURAL = Ø..MAXINT;     (* FOR USE IN CONFORMANT ARRAY DECLARATION *)

VAR
  A1 : ARRAY [1..1Ø] OF INTEGER;
  A2 : ARRAY [2..2Ø] OF INTEGER;

PROCEDURE DISPLAYIT(

    VAR AR1 : ARRAY [LOWBOUND..HIBOUND:NATURAL] OF INTEGER

                    );

(* THIS DECLARATION DEFINES THREE VARIABLES:

        AR1      : THE PASSED ARRAY
        LOWBOUND: THE LOWER BOUND OF AR1 (PASSED AT RUN-TIME)
        HIBOUND : THE UPPER BOUND OF AR1 (PASSED AT RUN-TIME)

*)

VAR
  I : NATURAL;
(* COMPATIBLE WITH THE INDEX TYPE OF THE CONFORMANT ARRAY *)

BEGIN
  FOR I := LOWBOUND TO HIBOUND DO
    WRITELN('INPUT ARRAY[',I,']=',AR1[I])
END;

BEGIN (* MAIN PROGRAM *)

  DISPLAYIT(A1);     (* CALL DISPLAYIT AND PASS A1 EXPLICITLY AND
                        1 AND 1Ø IMPLICITLY *)

  DISPLAYIT(A2)      (* CALL DISPLAYIT AND PASS A2 EXPLICITLY AND
                        2 AND 2Ø IMPLICITLY *)

END.
```

10.1 Standard procedures

The following is a list of Pascal/MT+ built-in procedures (except I/O which are listed in section 10.1.1). See the applications guide for parameters and usage. These procedures are pre-declared in a scope surrounding the program therefore any user routines of the same name will take precedence.

NEW	DISPOSE	EXIT	INSERT
DELETE	COPY	CONCAT	

10.1.1 File handling procedures

All standard file handling procedures are included. In addition the procedure ASSIGN(f,string) is added where f is a file and string is a literal or variable string. ASSIGN assigns the external file name contained in string to the file f. It is used preceeding a RESET or REWRITE. See section 5.16 in the Applications Guide for details.

Listed below are the names of the file handling procedures:

GET	PUT	RESET	REWRITE
ASSIGN	CLOSE	CLOSEDEL	PURGE
OPEN	OPENX	BLOCKREAD	BLOCKWRITE
CHAIN	PAGE		

10.1.2 Dynamic allocation procedures

In addition to NEW and DISPOSE, MEMAVAIL and MAXAVAIL are also included. See section 5.23 of the applications guide for a description of these functions.

10.1.3 Data transfer procedures

PACK UNPACK

11.0 Function declarations

```
<function decl>    ::= EXTERNAL <function heading> |
                       <function heading> <block>

<functon heading>  ::= FUNCTION <identifier> <parmlist> : <result type> ; |
                       FUNCTION <identifier> : <result type> ;

<result type>      ::= <type identifier>
```

11.1 Standard functions

Listed below are the names of the standard functions supported:

ABS	SQR	SIN	COS
EXP	LN	SQRT	ARCTAN
ODD	TRUNC	ROUND	ORD
WRD	CHR	SUCC	PRED
EOLN	EOF	IORESULT	MEMAVAIL
MAXAVAIL	ADDR	SIZEOF	POS
LENGTH	LENGTH		

11.1.1 Arithmetic functions

11.1.2 Predicates

11.1.3 Transfer functions

WRD(x) : The value x (a variable or expression) is treated as the WORD (unsigned integer) value of x.

11.1.4 Further standard functions

12.0 Input and Output

Pascal/MT+ supports all Standard Pascal I/O facilities.

In addition to the standard I/O facilities, Pascal/MT+ provides a mechanism by which Pascal/MT+ programmers can write their own character level I/O drivers in Pascal/MT+. This facility allows the ROM based program to be system independent and allows the user to use the input and output format conversion routines with strings, I/O ports, etc.

The re-directed I/O facility is simple and easy to use. The user must simply place the address of a routine, in square brackets, after the left parenthesis and before the parameter list in a READ, WRITE or WRITELN statement.

EXAMPLE:

```
READ( [ ADDR(getch) ], ...);

WRITELN( [ ADDR(putch) ], ... );
```

The "getch" and "putch" routines may be written in Pascal/MT+ or in assembly language. The parameter requirements for these routines are as follows:

```
FUNCTION getch : CHAR;

PROCEDURE putch( outputch: CHAR);
```

The declaration of these routines must be as shown. The names need not be getch/putch, but the parameters, none for getch and one for putch, must be exactly as shown, and the compiler does not check. The user may assign the address of the procedure to an integer using the ADDR function and then specify this integer (e.g. READ([P],...) which does not save execution time but does save typing time. Note that because EOLN and EOF require a file on which to operate READLN and EOF/EOLN cannot be used with re-directed I/O.

12.1 The procedure read

12.2 The procedure readln

```
<readcall>       ::= <read or readln> {( {<filevar> ,} {<varlist>} )}

<read or readln> ::= READ | READLN

<filevar>        ::= <variable>

<varlist>        ::= <variable> {,<variable>}
```

12.3 The procedure write

12.4 The procedre writeln

```
<writecall>        ::= <write or writeln> {( {<filevar> ,} {exprlist} )}

<write or writeln> ::= WRITE | WRITELN

<exprlist>         ::= <wexpr> {,<wexpr>}

<wexpr>            ::= <expression> {:<width expr> {:<dec expr>}}

<width expr>       ::= <expression>

<dec expr>         ::= <expression>
```

12.5 Additional procedures

See section 10.1.1

NOTES:

When reading or writing variables of type WORD the input is in HEX and the output is in HEX. When reading variables of type integer the user may force HEX input by preceeding the number with a '$' character. (e.g. $1F32)

13.0 Programs

```
<program>           :=  <program heading> <block> . |
                        <module  heading>
                            <label declaration part>
                            <constant definition part>
                            <type definition part>
                            <variable declaration part>
                            <procfunc declaration part>
                            MODEND .

<program heading>  ::= PROGRAM <identifier> ( <prog parms> ) ;

<module heading>   ::= MODULE <identifier> ;

<prog parms>       ::= <identifier> {,<identifier>}
```

Identical to the standard with the addition of modules see section 14.0.

14.0 Modular Compilation

Pascal/MT+ supports a flexible modular compilation system. Unlike other systems used for Pascal, such as UNITs, the Pascal/MT+ system allows an easy transition from large monolithic programming to modular programming without a great deal of pre-planning. Program may be developed in a monolithic fashion until they become too large to manage (or compile) and then split into modules at that time. The Pascal/MT+ modular compilation system allows full access to procedures and variables in any module from any other module. A compiler toggle is provided to allow the user to "hide" (i.e. make private) any group of variables or procedures. See section 2.5 in the applications guide for a discussion of the $E toggle.

The structure of a module is similar to that of a program. It begins with the reserved word MODULE followed by an identifier and semi- colon(e.g. MODULE TEST1;) and ends with the reserved word MODENI followed by a dot (e.g. MODEND.). In between these two lines the programmer may declare label, const, type, var, procedure and function sections just as in a program. Unlike a program, however, there is no BEGIN..END section after the procedure and function declarations, just the word MODEND followed by a dot (.) .

Example:

```
MODULE MOD1;

<label, const, type, var declarations>

<procedure / function declarations and bodies>

MODEND.
```

In order to access variables, procedures and functions in other modules (or in the main program) a new reserved word, EXTERNAL, has been added and is used for two purposes.

First, the word EXTERNAL may be placed after the colon and before the type in a GLOBAL variable declaration denoting that this variable list is not actually to be allocated in this module but is really in another module. No storage is allocated for variables declared in this way.

Example:

```
I,J,K : EXTERNAL INTEGER; (* in another module *)

R:      EXTERNAL RECORD   (* again in another module *)
                  ...      (* some fields *)
                END;
```

Note that the Pascal/MT+ system requires that the user be responsible for matching declarations identically as the compiler and linker do not have the ability to type check.

Second, the EXTERNAL word is used to declare procedures and functions which exist in other modules. These declarations must appear before the first normal procedure or function declaration in the module/program.

Note, just as in variable declaration the Pascal/MT+ system requires that the user make sure that the number and type of parameters match exactly and that the returned type match exactly for functions as the compiler and linker do not have the ability to type check across modules.

The user should note that in Pascal/MT+ external names are significant only to seven characters and not eight. When interfacing to assembly language the user must limit the length of identifiers accessable by assembly language to six characters (see section 13.2 of the applications guide for more information on external identifier naming conventions).

Listed below are a main program skeleton and a module skeleton. The main program references variables and subprograms in the module and the module references variables and subprograms in the main program. The only differences between a main program and a module are that at the beginning of a main program there are sixteen bytes of header code and a main program body following the procedures and functions.

Main Program Example:

```
PROGRAM EXTERNAL_DEMO;

<label, constant, type declarations>

VAR

   I,J : INTEGER;          (* AVAILABLE IN OTHER MODULES *)

   K,L : EXTERNAL INTEGER; (* LOCATED ELSEWHERE *)

EXTERNAL PROCEDURE SORT(VAR Q:LIST; LEN:INTEGER);

EXTERNAL FUNCTION  IOTEST:INTEGER;

PROCEDURE PROC1;
BEGIN
  IF IOTEST = 1 THEN
     (* CALL AN EXTERNAL FUNC NORMALLY *)
    ...
END;

BEGIN
  SORT(....)
  (* CALL AN EXTERNAL PROC NORMALLY *)
END.
```

Module Example: (Note these are separate files)

```
MODULE MODULE_DEMO;
< label, const, type declarations>
VAR
  I,J : EXTERNAL INTEGER;      (* USE THOSE FROM MAIN PROGRAM *)
  K,L : INTEGER;               (* DEFINE THESE HERE *)

EXTERNAL PROCEDURE PROC1;      (* USE THE ONE FROM THE MAIN PROG *)
PROCEDURE SORT(...);           (* DEFINE SORT HERE *)
    ...
FUNCTION IOTEST:INTEGER;       (* DEFINE IOTEST HERE *)
    ...
<maybe other procedures and functions here>
MODEND.
```

Pascal/MT+ Applications Guide

Pascal/MT+ APPLICATIONS GUIDE:

Table of Contents

1.0 Introduction

1.1 Purpose of Applications Guide

The Pascal/MT+ system is a complex series of programs, modules and run-time library subroutines. This applications guide is intended to help the Pascal/MT+ user to understand how to use the features of Pascal/MT+. The applications guide contains information on how to operate the compiler, linker, debugger and disassembler; a description of the implementation of Pascal/MT+ data types; a summary of built-in features and examples of their usage; run-time considerations including interfacing with other languages; and a list of the compiler error messages with the most common cause for each message.

1.2 Compile and run a sample program

Before compiling and running the sample program described in this section be sure that you have made a backup of all of the disks included with this software release.

The following is a step-by-step guide to the basic operation of the Pascal/MT+ system. You will compile, link and execute a sample program under the CP/M operating system. NOTE: If the Pascal/MT+ system you have purchased generates code for other than 8080/Z80 type CPUs then refer to the CPU applications guide for further information regarding the execution of programs on the target CPU.

The following discussion assumes that the computer on which you are about to execute Pascal/MT+ has two 8" floppy disks. If you have other than this configuration then make the appropriate adjustments. Please read all the documentation before attempting to operate the software so that you have an idea of what is being done.

STEP ONE: Put a CP/M system on your COPY of the distribution disk with the compiler on it or transfer the files from the distribution disk to your system disk.

STEP TWO: Place the disk now containing the compiler and CP/M into your 'A:' drive.

STEP THREE: Place your COPY of the sample programs diskette into your 'B:' drive.

STEP FOUR: Boot your system and remain logged into the 'A:' drive.

STEP FIVE: Type the following command (<cr> signifies you typing the return key on your system keyboard) :

```
MTPLUS B:CALC<cr>
```

STEP SIX: The compiler should load and display the message 'Pascal/MT+ 5.xx' where 'xx' is the sub-release number for the version of the software which you have. The compiler should process the CALC program by displaying the following: (or something close, we reserve the right to change without reprinting all the manuals).

```
Pascal/MT+ 5.xx

Code Gen:80
+++++
Source lines:      87

Phase 1
Available Memory:  nnnnn
User Table Space:  nnnnn   {after pre-defined symbols}
####
Remaining Memory:  nnnnn
Phase 2
8080
SUBREAL
ADDREAL
TF
CALC
CALCULAT

Lines :      87
Errors:       0
Code  :    1734
Data  :      42
Compilation Completed
```

STEP SEVEN: After the compilation is complete verify that the compiler properly placed the CALC.ERL file on the destination disk by typing 'DIR B:CALC.ERL' and having the CP/M system display:

```
                    B: CALC    ERL
```

STEP EIGHT: Now to link the program! Type:

```
LINKMT B:CALC,B:TRANCEND,B:FPREALS,B:PASLIB/S
```

followed by the return key. You should see the following output:

```
LINK/MT+ 5.00

Processing file-  B:CALC    .ERL
Processing file-  B:TRANCEND.ERL
```

```
Processing file-  B:FPREALS .ERL
Processing file-  B:PASLIB  .ERL

Undefined Symbols:

No Undefined Symbols


nnnn (decimal) records written to CALC    .COM

Total Data:  nnnnH bytes
Total Code:  nnnnH bytes
Remaining :  nnnnH bytes

Link/MT+ processing completed
```

STEP NINE: Now verify that the linker placed the CALC.COM file on the destination disk by typing: 'DIR B:CALC.COM' and receiving the response:

```
B: CALC    COM
```

from CP/M.

STEP TEN: Now to run the program! Type 'B:CALC' and you should be greeted with the message: 'ENTER FIRST OPERAND? '. Respond with '5.5' and <return>. Then the message 'R1= .5500000E+01' should appear followed by 'ENTER SECOND OPERAND? '. Respond with '99.256' followed by <return>. Then the message 'R2= .9925601E+02' should appear followed by 'ENTER OPERATOR:' followed by a list of operators. Respond with '+' followed by <return> Finally the result, '104.756' should be printed followed by 'TYPE <ESCAPE> TO STOP'. Type <escape> and that's it!.

1.3 Contents of Distribution disk

MTPLUS.COM	(compiler)
MT1????.OVL	
MT2????.OVL	
MT3????.OVL	
MT4????.OVL	(overlays for the ???? CPU)
LINKMT.COM	(linker)
PASLIB.ERL	(Run-time library object)
FPREALS.ERL	(Floating point REAL routines)
TRANCEND.ERL	(Floating point transcendental routines)
BCDREALS.ERL	(BCD Fixed point REAL routines)
DIS????.COM	(Disassembler)
PATCHER.COM	(Patch application program)
DEBUGGER.ERL	(symbolic debugger library)

Additional files are present on the disks. Consult the applications note which accompanies the software. A number of example programs, the source for the run-time library and other support tools such as the disassembler, etc. are supplied with various configurations of the system.

Note: The Pascal/MT+ system uses the extension .ERL for Extended ReLocatable files. These are, for the most part, fully compatible Microsoft relocatable format (for 8080/Z80 CPUs) but may contain extended record formats if the disassembler is being used. See section 3 and 10 of the applications guide.

2.0 Compiler operation

The compiler is named MTPLUS.COM and uses four overlays. Input files may be located on any disk and the names are arbitrary. The file may have any extension but if specified with a blank extension (e.g. TEST1) and not found with a blank extension then the compiler will search for a file with a .SRC extension followed by searching for a file with a .PAS extension. If no match is made then an error message will be issued: 'Unable to open input file'. MTPLUS creates a relocatable file <name>.ERL which must be linked with LINK/MT+ to the routines in the runtime package. See section 3 for details regarding linking.

The compiler accepts a number of "option switches" following the name of the input file on the command line. These options switches are in the form of a string preceeded by a '$' (dollar sign) character and are single letters followed by zero or more parameter characters. The parameter string extends from the $ to the end of the line and spaces are ignored (i.e. $PXRB is the same as $PX RB). They are listed below:

Compiler switch	Meaning
Rd	Put the .ERL file on 'd:'
nd	The .OVL file #n (n=1..4) is on 'd:'
Pd	Put the .PRN file on 'd:'
X	Generate an eXtended REL file including disassembler records
D	Generate debugger information in the object code and write the .PSY file to the drive specified by the R option
Ed	The MTERRS.TXT file is on 'd:'
Td	Put the PASTEMP.TOK file on 'd:'
Q	Quiet, suppress any unnecessary console messages
C	Continue on error, default is to pause and let operator interact on each error, one at a time.
A	Automatically call the linker at the end of compilation and link the .ERL file with the standard library only. The .COM file will be placed on the same disk as the .ERL file
B	Use BCD rather than floating point for the real numbers
Z	Generate Z80 optimized code (for 8080/Z80 version only)

Where a drive number is shown for the .PRN file the user ma specify 'X' which will cause the .PRN file to be displayed on th console. An example which executes the compiler, reads the sourc from the A: drive, places the .ERL file on B:, the .PRN file on th console and automatically calls the linker is as follows:

```
MTPLUS A:TESTPROG $RB PX A
```

The defaults for the compiler switches are:

R	.ERL file on same disk as source file
1..4	.OVL files are on the default disk
P	no .PRN file
X	non-extended file generated
D	no debugger information in object file and no .PSY file written
E	MTERRS.TXT on default disk
T	PASTEMP.TOK on default disk
Q	Compiler is verbose
C	Compiler stops and asks on each error
A	Compiler does not automatically chain to linker
B	Floating point reals are the default
Z	Generate 8080-only code (for 8080/Z80 version on

Various versions of the compiler will have a mechanism fo changing these default versions by a patch or a setup program Consult any applications notes which came with your package for mor information.

2.1 System requirements for running Pascal/MT+

The Pascal/MT+ system requires a 8080 or Z80 CPU running the CP/M operating system in which to operate. Other versions may execute with other operating systems and / or CPUs in the future, please consult any applications notes which came with your package for further details.

In a CP/M environment the minimum requirement is 140K of simultaneous on-line storage (i.e. the equivalent of two 5.25 in. mini-floppy disks). The design goal for Pascal/MT+ is that it will operate in a CP/M system with a minimum of 44K of Transient Program Area (TPA). (this is typically available in a 48K CP/M system). It is suggested that a minimum workable system for larger programs include at least 300K bytes of floppy disk and 52K of TPA.

2.2 Run-time requirements for Pascal/MT+

The Pascal/MT+ system generates programs which utilize a variety of run-time support subroutines which are extracted from PASLIB. These run-time routines handle such needs as multiply and divide on those processors which do not have such hardware and file input and output interface to the operating system.

For programs which are run under the CP/M operating system the minimum run-time overhead is typically in the 2K to 3K byte range. This includes support routines and text file I/O routines for integer, characters and strings. Additional modules will be included for routines which utilize REAL numbers, non-text file I/O, transcendental routines, etc.

For programs which are run in a stand-alone manner the user is required to write console/file I/O drivers for the target system. Complete source for the run-time library subroutines is provided and the applications note which accompanies the system describes the implementation of the I/O drivers for the system in question. The user should refer to section 12.4 for examples of how to create stand alone systems.

2.3 Invocation

To execute the Pascal/MT+ compiler type:

MTPLUS <filename> {optional parameters preceeded by $}

EXAMPLE:

```
MTPLUS CALC       {output CALC.ERL to default drive}.

MTPLUS CALC $RB   {output CALC.ERL to drive B:}
```

See section 2.Ø above for more information about the compiler options.

2.4 Compilation data

The Pascal/MT+ compiler will output a number of messages and characters during the compilation. For users who often wonder what is happening the Pascal/MT+ compiler will periodically output characters during the first two phases of the compilation (Phase Ø and Phase 1) to keep the user happy knowing that the compiler has not gone off to meet its maker.

A '+' is put out to the console for every 16 source code lines syntax scanned during Phase Ø. At the beginning of PHASE 1 the available memory space is displayed. This is the number of bytes (in decimal) of memory before generation of the symbol table. Approximately 3K worth of symbol table space is consumed by pre-defined identifiers. See section 2.5 on reducing this space by eliminating unneeded declarations of built-in routines. When a procedure or function is found a '#' is output to the console. At the completion of PHASE 1 the number of bytes remaining in memory is displayed in decimal.

PHASE 2 generates object code. When the body of each procedure is encountered the name of the procedure is output so that the user can see where the compiler is in the compilation of the program. Pascal/MT users will note that the compiler does not put the absolu addresses of the procedures out at compile time but the relative addresses for this module. The linker /M (Map) option will list the absolute addresses of the procedures in each module. Upon completion the following lines are displayed:

```
Lines :    Lines of source code compiled (in decimal).
Errors:    number of errors detected.
Code  :    bytes of code generated (in decimal).
Data  :    bytes of data reserved (in decimal).
```

2.5 Compiler toggles

The compiler toggle signals the compiler that the user wishes to enable or disable certain options. The format of this toggle is (*$____*) or {$_____} where the blanks are filled in with the toggle. The compiler does not accept blanks before the key letter or trailing or imbedded blanks in names but will skip over leading blanks (e.g. {$E +} is the same as {$E+}, but {$ E +} will be ignored).

EXAMPLES:

```
(*$E+*)
{$P}
{$I D:USERFILE.LIB}
```

$E+ and $E- controls the generation of entry point records in the relocatable file. $E+ causes the global variables and all procedures and functions to be available as entry points (i.e. available to be referenced by EXTERNAL declarations in other modules). $E- supresses the generation of these records thus causing the variables, procedures, and functions to be logically private. The default state is $E+ and the toggle may be turned on and off at will.

$S+ enables stack frame allocation of procedure / function parameters and local variables. This must be turned on before the word PROGRAM or MODULE and, unlike Pascal/MT, cannot be turned off within a separately compiled unit. Global variables in either programs or modules are always allocated statically. Modules which use $S+ may be mixed with modules which do not.

$I<filename> causes the compiler to include the named file in the sequence of Pascal source statements. Filename specification includes drive name and extension in CP/M standard format.

The $Z nnnn toggle is used to initialize the stack pointer to nnnnH in non-CP/M environments. In a CP/M environment the hardware stack is initialized by loading the value in absolute location 0006 into the stack pointer register. If the $Z toggle is used then generation of the CP/M type initialization is supressed.

$T+, $T-, $W+ and $W- control the strict type checking / non-portable warning facility. These features are tightly coupled (i.e. strict type checking implies warning non-portable usage and visa versa). The default state is $T- ($W-) in which type checking is relaxed and warning messages are not generated. This may be turned on and off throughout the source code as desired.

$R+ and $R- control the compiler's generation of run-time code which will perform range checking on array subscripting and storing into subrange variables. The default state is $R- (off) and this toggle may be turned on and off throughout the source code as desired.

$X+ and $X- control the compiler's generation of run-time code which will perform run-time error checking and error handling for what is termed eXceptions. Exceptions are:

```
Zero divide
String overflow / truncation
Heap overflow
```

The system philosophy under which Pascal/MT+ operates states zero divide and string overflow are treated in a "reasonable" manner when exception checking is disabled. Zero divide returns the maximum value for the data type and string overflow results in truncation of the string rather than modification of adjacent memory areas. The default state is $X- and may be changed throughout the source code as desired. The user is directed to section 14 for more discussion of run-time error handling and options.

The $P and $L+, $L- toggles control the listing generated by the first pass of the compiler. $P will cause a formfeed character (CHR(12)) to be inserted into the .PRN file. $L+ and $L- are used to switch the listing on and off throughout the source program and may be placed wherever desired.

The $Cn toggle can be used by the user to reduce run-time object code memory requirements when using REAL arithmetic. The user can, if available, specify a restart instruction number and the compiler will then change all calls to the @XOP routine (see section 12.1) into a restart instruction. This will cause all 3 bytes call instructions to shrink to one byte call instructions. The user specifies 'n' in the range 0..7 and the compiler generates RST n instructions. In a CP/M environment the restarts which are not available because of CP/M usage are: 0 and 7. MP/M users and others should consult their hardware documentation for more details. This facility is available only in the 8080/Z80 systems (using restarts). Similar facilities may be available in other CPU systems. Consult the appropriate CPU applications guide for details.

The $Kn toggles are used to remove unneeded built-in routine definition from the symbol table to make more room for user symbols. The value n (Ø..6) is used to control various groups of routines. These may be used in any combination but these toggles MUST appear before the word PROGRAM or MODULE to be effective. The value n is selected as follows:

Group	Routines Removed
Ø	ROUND, TRUNC, EXP, LN, ARCTAN SQRT, COS, SIN
1	COPY, INSERT, POS, DELETE, LENGTH CONCAT
2	GNB, WNB, CLOSEDEL, OPENX, BLOCKREAD BLOCKWRITE
3	CLOSE, OPEN, PURGE, CHAIN, CREATE
4	WRD, HI, LO, SWAP, ADDR, SIZEOF, INLINE, EXIT, PACK, UNPACK
5	IORESULT, PAGE, NEW, DISPOSE
6	SUCC, PRED, EOF, EOLN
7	TSTBIT, CLRBIT, SETBIT, SHR, SHL

THE USER SHOULD NOTE THAT THIS ONLY REMOVES THE NAMES FROM THE PRE-DEFINED SYMBOL TABLE TO MAKE ROOM FOR USER SYMBOLS. THESE ROUTINES ARE ONLY INCLUDED IN THE USER'S PROGRAM BY THE LINKER IF THEY ARE USED IN THE PROGRAM.

Listed below is a summary of available compiler toggles

Compiler Toggles		Default
$E +/-	Controls entry point generation	$E+
$S +/-	Controls recursive/static variables	$S-
$I <name>	Includes another source file into the input stream (e.g. {$I XXX.LIB})	
$R +/-	Controls range checking code	$R-
$T +/- $W +/-	Controls strict type checking and generation of warning messages	$T- $W-
$X +/-	Controls eXception checking code	$X-
.$P	Enter a formfeed in the .PRN file	
$L +/-	Controls the listing of source code	$L+
$Kn	Allows for Killing built-in routines to save space in symbol table (n=Ø..7)	
$Z nnnn	Initialize hardware stack to nnnnH (default is contents of location ØØØ6 at the begining of execution)	
$Cn	Use RST n instructions for REAL operations (default is to use CALL instructions)	

2.6 Error messages

Compilation errors are numbers which have the same meaning as those in Jensen and Wirth's 'User Manual and Report'. The errors messages, brief explanations, and some causes of the error are found in the appendix.

3.0 Linker operation

3.1 Invocation and commands

LINK/MT+ is used by typing its name followed by a space followed by the main program and modules to be linked separated by commas. The output is directed to the same disk as the main program unless the user specifies an output file name followed by an equal sign before the main program name. Examples are shown below:

EXAMPLE:

```
LINKMT CALC,TRANCEND,FPREALS,PASLIB/S

LINKMT B:CALC=CALC,TRANCEND,FPREALS PASLIB/S    {CALC.COM is put to B:}
```

The above command will link one of the demo programs with the run time package. The items to be linked may be preceeded by a drive letter:

```
LINKMT A:CALC,D:TRANCEND,F:FPREALS,B:PASLIB/S
```

The linker allows the user to place a number of "switches" following the file names in the list. These switches are preceeded by a slash (/) and are a single letter with a parameter on the P and D switches.

The examples above show the use of the /S switch which informs the linker to search the module as a library and extract only the necessary routines. A /M following the last file named in the parameter list generates a map. A /L following the last module named causes the linker to display module code and data locations as they are being linked. A /E following the last module causes the linker to display all routines including those which begin with $, ? or @ which are reserved for run-time library routine names.

In order to support relocation of object code and data areas the linker supports the /P and /D switches. The /P switch controls the location of the object code (ROM) and the /D switch controls the location of the data areas (RAM). The syntax is: /P:nnnn or /D:nnnn where "nnnn" is a hexadecimal number in the range 0..FFFF.

Using the /D switch will also allow linking of larger programs because the data area is not reserved in memory during the linking operation. The user should note that local file operations are not guaranteed if this is used because the system depends upon the linker zeroing the data area to make this facility work properly.

Using the /P switch and /D switch does not cause the linker to

leave empty space at the beginning of the .COM file. Other linkers (in particular L80) will generate a significant ammount of disk space to force the program to load at the proper address in a CP/M environment. The philosophy of LINK/MT+ is that if the /P switch is used the user really wants to move the program to another system for execution. This means that if the user specifies /P:8000 that the first byte of the .COM file will be the byte to be placed at location 8000H and not 32K of zeroes before the first byte. In addition, if the user specifies /D the linker will not save any of the data area in the .COM file. This is a good way for reducing the data storage on disk for programs since only the code will be loaded from disk and not uninitialized data areas.

These switches (/P and /D) may be specified after the last routine to be loaded and may be in any order.

The /H:nnnn switch is provided to allow the linker to generate a .HEX file instead of a .COM file. The nnnn value is in HEX and is totally independent of the default relocation value of 100H (possibly overridden by the /P switch). This means that the user may relocate the program to execute at say 1D00H but generate the .HEX file to have addresses starting at 8000H. (the user would use /P:1D00/H:8000).

The user in a CP/M environment must typically use the SUBMIT facility for typing repetitive sequences such as linking multiple files together over and over and over again. The LINK/MT+ linker allows the user to enter this data into a file and have the linker process the file names from the file. This process is considerably faster than submit. The user must specify a file with an extension of .CMD and follow this file name with a /F (e.g. CFILES/F). The linker will read input from this file and process the names. The input from the file is concatenated logically between the data on the left of the file name and the data on the right of the /F switch. The total input buffer is 256 bytes.

Listed below is a summary of the switches:

Linker Switch Summary

/S	-	Search preceeding name as a library extracting only the required routines
/L	-	List modules as they are being linked
/M	-	List all entry points in tabular form
/E	-	List entry points beginning with $, ? or @ in addition to other entry points
/P:nnnn	-	Relocate object code to nnnnH
/D:nnnn	-	Relocate data area to nnnnH
/W	-	Write a SID compatible .SYM file (written to the same disk as the .COM file)
/H:nnnn	-	Write the output as a .HEX file with nnnnH as the starting location for the hex format. This is totally independent of the /P switch (no .COM file produced if this switch is used)
/F	-	Take preceeding file name as a .CMD file containing file names (one per line)

The linker will take up to thirty two names on the command line (or command file input) for files to be linked.

Errors encountered in the linking process are self explanatory such as 'unable to open input file: xxxxxxxx' and 'Duplicate symbol - xxxxxxx'.

3.2 Attributes of linkable modules

Link/MT+ will link Pascal/MT+ main programs, Pascal/MT+ modules, and assembly language modules created by M80 or RMAC. Link/MT+ supports those features of the Microsoft relocatable format required for Pascal/MT+. These do not include: External plus offset, External minus offset, COMMON, initialized DATA areas in the DATA segment, and request library search. Also Link/MT+ demands that the data size and program size records preceed the first byte of data to be loaded. This is the case with the Pascal/MT+ compiler, M80 and RMAC but not with such compilers as FORTRAN. MT MicroSYSTEMS recommends using the linker supplied with the other language processor be used if mixed linking of Pascal and alien modules (other than assembly language) is to be performed.

3.3 Using other linkers

If the user has not specified that the disassembler is to be used then the .ERL file produced by the Pascal/MT+ compiler is totally Microsoft compatible. As shown in section 15 linking with other languages such as FORTRAN may be done using specially constructed routines which translate Pascal/MT+ parameter lists into FORTRAN parameter lists. Other linkers, particularly the L8Ø linker from Microsoft, may not be able to link a program which Link/MT+ can handle due to memory limitations imposed by the design of these other linkers.

4.0 Data Types

This section describes how the standard Pascal data types are implemented in Pascal/MT+. A summary of the data types appears in the following table.

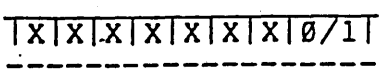
Data type	Size	Range
CHAR	1 8-bit-byte	0..255
BOOLEAN	1 8-bit-byte	false..true
INTEGER	1 8-bit-byte	0..255
INTEGER	2 8-bit-bytes	-32768..32767
BYTE	1 8-bit-byte	0..255
WORD	2 8-bit-bytes	0..65535
BCD REAL	10 8-bit-bytes	18 digits,4 decimal
FLOATING REAL	4 8-bit-bytes	10E-17..10E+17
STRING	1..256 bytes	----------
SET	32 8-bit-bytes	0..255

4.1 CHAR

The data type CHAR is implemented using one 8-bit byte for each character. The reserved word PACKED is assumed on arrays of CHAR. CHAR variables may have the range of CHR(0) .. CHR(255). When pushed on the stack a CHAR variable is 16 bits with the high order byte containing 00. This is to allow ORD, ODD, CHR and WRD to all work together.

4.2 BOOLEAN

The data type BOOLEAN is implemented using one 8-bit byte for each BOOLEAN variable. When pushed on the stack ,8 bits of 0 are pushed to provide compatibility with built in operators and routines. The reserved word PACKED is allowed but does not compress the data structure any more than one byte per element (this occurs with and without the packed instruction). ORD(TRUE) = 0001 and ORD(FALSE) = 0000. The BOOLEAN operators AND, OR and NOT operate only on ONE byte. The user is refered to the &, ! and ~ operators (see section 8 of the language guide) for 16-bit boolean operators.



(X means don't care)

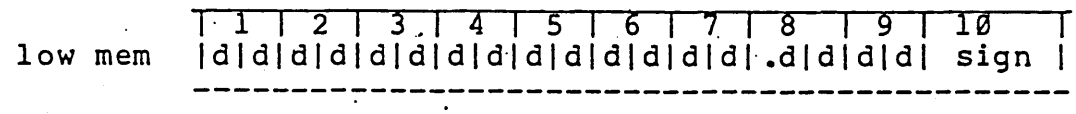
4.3 INTEGER

The data type INTEGER is implemented using two 8-bit bytes for each INTEGER variable. The order of the bytes is CPU dependent. In the 8080, 8085, Z80, 8086 and 8088 the low byte is in lower numbered address and the high order 8 bits are in the higher numbered address. In the 68000 and Z8000 the high byte is in the low numbered address and the low byte is in the higher numbered address. MAXINT = 32767 and INTEGERS can have the range -32768..32767. An integer subrange declared to be within the 0..255 occupies only one byte of memory instead of two bytes. Integer constants may be hexadecimal numbers by preceeding the hex number with a dollar sign (e.g. $0F3B).

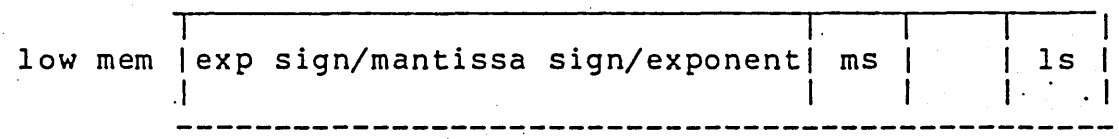
4.4 REAL

The implementation of the data type REAL in Pascal/MT+ has been done in two different ways to serve the needs of two different market areas.

For business applications the REAL data type has been implemented in BCD with 18 digits and 4 fixed decimal places. Automatic rounding is done after the fourth place during calculations and also at the specified place if formatted output is used. The format of a REAL BCD number is: bytes 1..9 are digits packed two to the byte, and byte 1Ø contains the sign: Ø for positive and $FF for negative.



For scientific and engineering applications the REAL data type has been implemented using binary floating point. The floating point used in Pascal/MT+ is fully compatible with the AMD 9511 hardware floating point unit (also being second sourced by Intel). Thirty-two (32) bits (4-bytes) of data are required to implement a floating point number. The first byte contains the mantissa sign, the exponent sign and the exponent. The remaining three bytes contain the mantissa. The precision of this format is approximately 6.5 digits. The reader is referred to the AMD 9511 hardware manual for further details regarding the binary format.



ms = most significant bits
ls = least significant bits

Pascal/MT+ implements this floating point data type in both software and hardware. The standard floating point package system comes with software run-time. The source for the run-time package is used to modify port addresses for the 9511 to adapt this version of the run-time package to the user's system. The equate HARDWARE is used to control inclusion of the desired floating point routines into the run-time package.

4.5 Byte

The BYTE data type occupies a single byte. It is compatible with both INTEGER and CHAR types. This can be very useful when manipulating control characters, handling character arithmetic, etc. Characters and integers may be assigned to a BYTE.

4.6 Word

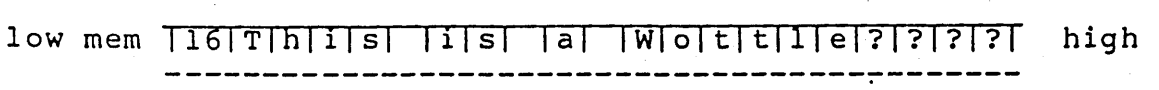
WORD is an unsigned native machine word. All arithmetic performed on expressions of type WORD is unsigned. In addition all comparisons are also unsigned. The WORD data type is designed such that it is always the same size as pointers. This particularly important in the 16-bit CPUs in which the integer size is 16-bits but typically the pointer size is 32-bits.

4.7 String

4.7.1 Definition

The pre-declared type STRING is like a packed array of characters in which the byte Ø contains the dynamic length of the string and bytes 1 through n contain the characters. Strings may be up to 255 characters in length. The default length is 8Ø characters which may be altered when a variable of type string is declared (see example below).

The string "This is a Wottle" is sixteen characters in length. The following diagram shows how these characters are stored in a string declared to be 2Ø characters in length.



If the number of characters in the string is less than the declared length, those bytes on the end are not defined. Note that the length is stored in the first byte and the total number of bytes required for the string is 17.

EXAMPLE:

```
VAR
  LONG_STR:        STRING;        {This may contain up to 80 characters}
  SHORT_STR:       STRING[10];    {This may contain up to 10 characters}
  VERY_LONG_STR :  STRING[255];   {This may contain up to 255 characters,
                                   the maximum allowed. }
```

4.7.2 Assignment

Assignment to a string variable may be made via the assignment statement, reading into a string variable using READ or READLN, or the pre-defined string functions and procedures.

EXAMPLE:

```
PROCEDURE ASSIGN;
VAR
  LONG_STR  : STRING;
  SHORT_STR : STRING[12];
BEGIN

  LONG_STR := 'This string may contain as many as eighty characters';
  WRITELN(LONG_STR);

  WRITE('type in a string 10 characters or less : ');
  READLN(SHORT_STR);
  WRITELN(SHORT_STR);

  SHORT_STR := COPY(LONG_STR,1,11);
  WRITELN('COPY(LONG_STR..)=',SHORT_STR);
END;
```

Output:

```
This string may contain as many as eighty characters
type in a string 10 characters or less : {123456} (USER INPUT)
123456
COPY(LONG_STR..)=This string m
```

Individual characters in a string variable are accessed as if the string is an array of characters. Thus, normal array subscripting via constants, variables, and expressions allows assignment and access to individual bytes within the string. Access to the string over its entire declared length is legal and does not cause a run-time error even if an access is made to a portion of the string which is beyond the current dynamic length. If the string is actually 20 characters long and the declared length is 30 then STRING[25] is accessible.

EXAMPLE

```
PROCEDURE ACCESS;
VAR
  I : INTEGER;
BEGIN
  I := 15;
  LONG_STR := '123456789abcdef';
  WRITELN(LONG_STR);
  WRITELN(LONG_STR[6], LONG_STR[ i-5 ]);
  LONG_STR[16] := '*';
  WRITELN(LONG_STR[16]);
  WRITELN(LONG_STR); (* will still only write 15-characters *)
END;
```

Output:

```
123456789abcdef
6a
*
123456789abcdef
```

4.7.3 Comparisons

Comparisons are valid between two variables of type string (regardless of their length) or between a variable and a literal string. Literal strings are sequences of characters between single quote marks. Comparisons may also be made between a string and a character. The compiler 'forces' the character to become a string by using the CONCAT buffer, therefore comparison of the result of the CONCAT function and a character is not meaningful as this would result in an always equal comparison.

EXAMPLE

```
PROCEDURE COMPARE;

VAR
  S1,S2 : STRING[10];
  CH1   : CHAR;

BEGIN
  S1 := '012345678';
  S1 := '222345678';

  IF S1 < S1 THEN
    WRITELN(S1,' is less than ',s2);

  S1 := 'alpha beta';
  IF S1 = 'alpha beta  ' THEN
    WRITELN('trailing blanks dont matter')
  ELSE
    WRITELN('trailing blanks count');
  IF S1 = '    alpha beta' THEN
    WRITELN('blanks in front don''t matter')
  ELSE
    WRITELN('blanks in front do matter');
  IF S1 = 'alpha beta' THEN
    WRITELN(S1,' = ',S1);
  S1 := 'Z';
  CH1 := 'Z';
  IF S1 = CH1 THEN
    WRITELN('strings and chars may be compared');
END;
```

Output:

```
012345678 < 222345678
trailing blanks don't matter
blanks in front do matter
alpha beta = alpha beta
strings and chars may be compared
```

4.8 SET

The SET data type is always stored as a 32 byte item. Each element of the set is stored as one bit. The low order bit of each byte is the first bit in that byte of the set. Shown below is the set 'A'..'Z' (bits 65..122)

Byte number	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	...	1F
Contents	00	00	00	00	00	00	00	00	FE	FF	FF	07	00	...	00

5.0 Summary of built-in procedures and parameters

This section provides descriptions and examples of Pascal/MT+ built-in procedures and functions. Each routine is described syntactically followed by a description of the parameters and an example program using the procedure or function. Section 5.24 provides a quick reference summary of all the built-in procedures and functions.

5.1 MOVE,MOVERIGHT, MOVELEFT

```
PROCEDURE MOVE      (SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVELEFT  (SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVERIGHT(SOURCE, DESTINATION, NUM_BYTES)
```

These procedures move the number of bytes contained in NUM_BYTES from the location named in SOURCE to the location named in DESTINATION. MOVELEFT moves from the left end of the source to the left end of the destination. MOVE is a synonym for MOVELEFT. MOVERIGHT moves from the right end of the source to the right end of the destination (the parameters passed to MOVERIGHT specify the left hand end of the source and destination).

The source and destination may be any type of variable and both need not be of the same type. These may also be pointers to variables or integers used as pointers. They may not be named or literal constants. The number of bytes is an integer expression greater than 0.

Watch out for these problems: 1) Since no checking is performed as to whether the number of bytes is greater than the size of the destination, spilling over into the data storage adjacent to the destination will occur if the destination is not large enough to hold the number of bytes; 2) Moving 0 bytes moves nothing; 3) No type checking is done; 'Along with freedom comes responsibility'.

MOVELEFT and MOVERIGHT are used to transfer bytes from one data structure to another or to move data around within a single data structure. The move is done on a byte level so the data structure type is ignored. MOVERIGHT is useful for transfering bytes from the low end of an array to the high end. Without this procedure a FOR loop would be required to pick up each character and put it down at a higher address. MOVERIGHT is also much, much faster. MOVERIGHT is ideal to use in an insert character routine whose purpose is to make room for characters in a buffer.

MOVELEFT is useful for transferring bytes from one array to another, deleting characters from a buffer, or moving the values in one data structure to another.

EXAMPLE:

```
PROCEDURE MOVE_DEMO;
CONST
  STRINGSZ = 80;
VAR
  BUFFER : STRING[STRINGSZ];
  LINE : STRING;

PROCEDURE INSRT(VAR DEST : STRING; INDEX : INTEGER; VAR SOURCE : STRING);
BEGIN
  IF LENGTH(SOURCE) <= STRINGSZ - LENGTH(DEST) THEN
    BEGIN
      MOVERIGHT(DEST[ INDEX ], DEST[ INDEX+LENGTH(SOURCE) ],
              LENGTH(DEST)-INDEX+1);
      MOVELEFT(SOURCE[1], DEST[INDEX], LENGTH(SOURCE));
      DEST[0] :=CHR(ORD(DEST[0]) + LENGTH(SOURCE))
    END;
END;

BEGIN
  WRITELN('MOVE_DEMO......');
  BUFFER := 'Judy J. Smith/  335 Drive/ Lovely, Ca.  95666';
  WRITELN(BUFFER);
  LINE := 'Roland ';
  INSRT(BUFFER, POS('5',BUFFER)+2,LINE);
  WRITELN(BUFFER);
END;
```

THE OUTPUT FROM THIS PROCEDURE:

```
MOVE_DEMO......
Judy J. Smith/  355 Drive/ Lovely, Ca.  95666
Judy J. Smith/  355 Roland Drive/ Lovely, Ca.  95666
```

5.2 EXIT

```
PROCEDURE EXIT;
```

Procedure EXIT will exit the current procedure/function or main program. EXIT will also load the registers and re-enable interrupts before exiting if EXIT is used in an INTERRUPT procedure. EXIT is the equivalent of the RETURN statement in FORTRAN or BASIC. It is usually executed as a statement following a test.

EXAMPLE:

```
PROCEDURE EXITTEST;
{ EXIT THE CURRENT FUNCTION OR MAIN PROGRAM. }

  PROCEDURE EXITPROC(BOOL : BOOLEAN);

  BEGIN
    IF BOOL THEN
      BEGIN
       WRITELN('EXITING EXITPROC');
       EXIT;
      END;
    WRITELN('STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY');
  END;

BEGIN
  WRITELN('EXITTEST.......');
  EXITPROC(TRUE);
  WRITELN('IN EXITTEST AFTER 1ST CALL TO EXITPROC');
  EXITPROC(FALSE);
  WRITELN('IN EXITTEST AFTER 2ND CALL TO EXITPROC');
  EXIT;
  WRITELN('THIS LINE WILL NEVER BE PRINTER');
END;

Output:
EXITTEST.......
EXITING EXITPROC
IN EXITTEST AFTER 1ST CALL TO EXITPROC
STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY
IN EXITTEST AFTER 2ND CALL TO EXITPROC
```

5.3 TSTBIT, SETBIT, CLRBIT

```
FUNCTION  TSTBIT(    BASIC_VAR, BIT_NUM) : BOOLEAN;
PROCEDURE SETBIT(VAR BASIC_VAR, BIT_NUM);
PROCEDURE CLRBIT(VAR BASIC_VAR, BIT_NUM);
```

BASIC_VAR is any 8 or 16 bit variable such as integer, char, byte, word, or boolean. BIT_NUM is 0..15 with bit 0 on the right. Attempting to set bit 10 of an 8 bit variable does not cause an error but has no effect on the end result.

TSTBIT returns TRUE if the designated bit in the basic_var is on, and returns FALSE if the bit is off. SETBIT sets the designated bit in the parameter. CLRBIT clears the designated bit in the parameter.

These procedures are useful for generating wait loops or altering incoming data by flipping a bit where needed. Another application is in manipulating a bit mappped screen.

EXAMPLE:

```
PROCEDURE TST_SET_CLR_BITS;

VAR
  I : INTEGER;
BEGIN
  WRITELN('TST_SET_CLR_BITS.......');
  I := 0;
  SETBIT(I,5);
  IF I = 32 THEN
    IF TSTBIT(I,5) THEN
      WRITELN('I=',I);
  CLRBIT(I,5);
  IF I = 0 THEN
    IF NOT (TSTBIT(I,5)) THEN
      WRITELN('I=',I);
 end;

Output:
TST_SET_CLR_BITS.......
I=32
I=0
```

5.4 SHR, SHL

```
FUNCTION SHR(BASIC_VAR, NUM) : INTEGER;
FUNCTION SHL(BASIC_VAR, NUM) : INTEGER;
```

BASIC_VAR is an 8 or 16 bit variable. NUM is an integer expression. SHR shifts the BASIC_VAR by NUM bits to the right inserting Ø bits. SHL shifts the BASIC_VAR by NUM bits to the left inserting Ø bits.

The uses of SHR and SHL are generally obvious. Suppose a 1Ø bit value is to be obtained from two separate input ports. Use SHL to read them in:

```
X := SHL(INP[8] & $1F, 3) ! (INP[9] & $1F);
```

The above example reads from port # 8, masks out the three high bits returned from the INP array, and shifts the result left. Next, this result is logically OR'd with the input from port # 9 which has also been masked.

The following procedure demonstrates the expected result of executing these two functions.

EXAMPLE:

```
PROCEDURE SHIFT_DEMO;
VAR I : INTEGER;
BEGIN
  WRITELN('SHIFT_DEMO........');
  I := 4;
  WRITELN('I=',I);
  WRITELN('SHR(I,2)=',SHR(I,2));
  WRITELN('SHL(I,4)=',SHL(I,4));
end;

Output:
SHIFT_DEMO........
I=4
SHR(I,2)=1
SHL(I,4)=64
```

5.5 HI, LO, SWAP

```
FUNCTION   HI(BASIC_VAR) : INTEGER;
FUNCTION   LO(BASIC_VAR) : INTEGER;
FUNCTION SWAP(BASIC_VAR) : INTEGER;
```

HI returns the upper 8 bits of BASIC_VAR (an 8 or 16 bit variable) in the lower 8 bits of the result. LO returns the lower 8 bits with the upper 8 bits forced to zero. SWAP returns the upper 8 bits of basic_var in the lower 8 bits of the result and the lower 8 bits of basic_var in the upper 8 bits of the result. Passing an 8 bit variable to HI causes the result to be Ø and passing 8 bits to LO does nothing.

These functions enhance Pascal/MT+'s abilities to read and write to I/O ports. If a data item has 16 bits of information to send to a port which can handle 8 bits at a time, use LO and HI to send the low byte followed by the high byte. Similarly, reading 16 bits worth of data from a port which sends 8 bits at a time may be performed by SWAPing the first 8 bits into the high byte:

```
OUT[6] := LO(B);
OUT[6] := HI(B);
B := SWAP(INP[7]) ! INP[7];
```

The following example shows what the expected results of these functions should be:

EXAMPLE:

```
PROCEDURE HI_LO_SWAP;
VAR
  HL : INTEGER;
BEGIN
  WRITELN('HI_LO_SWAP.......');
  HL := $104;
  WRITELN('HL=',HL);
  IF HI(HL) = 1 THEN
    WRITELN('HI(HL)=',HI(HL));
  IF LO(HL) = 4 THEN
    WRITELN('LO(HL)=',LO(HL));
  IF SWAP(HL) = $0401 THEN
    WRITELN('SWAP(HL)=',SWAP(HL));
END;

Output:
HI(HL)=1
LO(HL)=4
SWAP(HL)=1025
```

5.6 ADDR

```
FUNCTION ADDR(VARIABLE REFERENCE) : INTEGER;
```

ADDR returns the address of the variable referenced. Variable reference includes procedure/function names, subscripted variables and record fields. It does not include named constants, user defined types, or any item which does not occupy code or data space.

This function is used to return the address of anything: compile time tables generated by INLINE, the address of a data structure to be used in a move statement, etc.

EXAMPLE:

```
PROCEDURE ADDR_DEMO(PARAM : INTEGER);
VAR
  REC : RECORD
         J : INTEGER;
         BOOL : BOOLEAN;
       END;
  ADDRESS : INTEGER;
  R : REAL;
  S1 : ARRAY[1..10] OF CHAR;

BEGIN
  WRITELN('ADDR_DEMO.....');
  WRITELN('ADDR(ADDR_DEMO)=',ADDR(ADDR_DEMO));
  WRITELN('ADDR(PARAM)=',ADDR(PARAM));
  WRITELN('ADDR(REC)=',ADDR(REC));
  WRITELN('ADDR(REC.J) ', ADDR(REC.J));
  WRITELN('ADDR(ADDRESS)=',ADDR(ADDRESS));
  WRITELN('ADDR(R)=',ADDR(R));
  WRITELN('ADDR(S1)=',ADDR(S1));
end;
```

Output is system dependent

5.7 WAIT

```
PROCEDURE WAIT(PORTNUM , MASK, POLARITY);
```

PORTNUM and MASK are literal or named constants. POLARITY is a boolean constant.

WAIT generates a tight status wait loop:

```
IN portnum
ANI mask
J?? $-4
```

where ?? is Z if polarity is false and is NZ if polarity is true.

EXAMPLE:

```
PROCEDURE WAIT_DEMO;
CONST
  CONSPORT = $F7; (* for EXO NOBUS-Z COMPUTER *)
  CONSMASK = $Ø1;

BEGIN
  WRITELN('WAIT_DEMO.......');
  WRITELN('WAITING FOR A CHARACTER');
  WAIT(CONSPORT,CONSMAXK,TRUE);
  WRITELN('THANKS!');
end;
```

5.8 SIZEOF

```
FUNCTION SIZEOF(VARIABLE OR TYPE NAME) : INTEGER;
```

Parameter may be any variable: character, array, record, etc, or any user defined type. SIZEOF returns the size of the parameter in bytes. It is used in move statements for the number of bytes to be moved. With SIZEOF the programmer does not need to keep changing constants as the program evolves:

EXAMPLE:

```
PROCEDURE SIZE_DEMO;
VAR
  B : ARRAY[1..10] OF CHAR;
  A : ARRAY[1..15] OF CHAR;
BEGIN
  WRITELN('SIZE DEMO.......');
  A := '**********';
  B := '0123456789';
  WRITELN('SIZEOF(A)=',SIZEOF(A),' SIZEOF(B)=',SIZEOF(B));
  MOVE(B,A,SIZEOF(B));
  WRITELN('A= ',A);
end;
```

Output:

```
SIZEOF(A)=15 SIZEOF(B)=10
0123456789*****
```

5.9 FILLCHAR

```
PROCEDURE FILLCHAR( DESTINATION, LENGTH, CHARACTER);
```

DESTINATION is a packed array of characters. It may be subscripted. LENGTH is an integer expression. CHARACTER is a literal or variable of type char. Fill the DESTINATION (a packed array of characters) with the number of CHARACTERs specified by LENGTH.

The purpose of FILLCHAR is to provide a fast method of filling in large data structures with the same data. For instance, blanking out buffers is done with FILLCHAR.

EXAMPLE:

```
PROCEDURE FILL_DEMO;
VAR
   BUFFER : PACKED ARRAY[1..256] OF CHAR;
BEGIN
  FILLCHAR(BUFFER,256,' ');              {BLANK THE BUFFER}
END;
```

5.10 LENGTH

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

Returns the integer value of the length of the string.

EXAMPLE:

```
PROCEDURE LENGTH_DEMO;
VAR
  S1 : STRING[40];
BEGIN
  S1 := 'This string is 33 characters long';
  WRITELN('LENGTH OF ',S1,'=',LENGTH(S1));
  WRITELN('LENGTH OF EMPTY STRING = ',LENGTH(''));
end;
```

Output:

```
LENGTH OF This string is 33 characters long=33
LENGTH OF EMPTY STRING = 0
```

5.11 CONCAT

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

Return a string in which all sources in the parameter list are concatenated. The sources may be string variables, string literals, or characters.

EXAMPLE:

```
PROCEDURE CONCAT_DEMO;
VAR
  S1,S2 : STRING;
BEGIN
  S1 := 'left link, right link';
  S2 := 'root root root';
  WRITELN(S1,'/',S2);
  s1 := CONCAT(S1,' ',S2,'!!!!!!');
  WRITELN(S1);
end;
```

Output:

```
left link, right link/root root root
left link, right link root root root!!!!!!
```

5.12 COPY

```
FUNCTION COPY( SOURCE, LOCATION, NUM_BYTES) :.STRING;
```

SOURCE must be a string. LOCATION and NUM_BYTES are integer expressions. Return a string which contains the number of characters specified in NUM_BYTES from SOURCE beginning at the index specified in LOCATION.

EXAMPLE:

```
PROCEDURE COPY_DEMO;
BEGIN
  LONG_STR := 'Hi from Cardiff-by-the-sea';
  WRITELN(COPY(LONG_STR,9,LENGTH(LONG_STR)-9+1));
end;
```

Output:

```
Cardiff-by-the-sea
```

5.13 POS

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

Return the integer value of the position of the first occurence of PATTERN in SOURCE. If the pattern is not found a zero is returned. SOURCE is a string and PATTERN is a string, a character, or a literal.

EXAMPLE:

```
PROCEDURE POS_DEMO;
VAR
  STR,PATTERN : STRING;
  CH : CHAR;
BEGIN
  STR := 'MT MicroSYSTEMS';
  PATTERN := 'croSY';
  CH := 'T';
  WRITELN('pos of ',PATTERN,' in ',STR,' is ', POS(PATTERN,STR));
  WRITELN('pos of ',CH,' in ',STR,' is ',POS(CH,STR));
  WRITELN('pos of ''z'' in ',STR,' is ',POS('z',STR));
end;

Output:
pos of croSY in MT MicroSYSTEMS is 6
pos of T in MT MicroSYSTEMS is 2
pos of 'z' in MT MicroSYSTEMS is 0
```

5.14 DELETE

```
PROCEDURE DELETE( TARGET, INDEX, SIZE);
```

TARGET is a string. INDEX and SIZE are integer expressions. Remove SIZE characters from TARGET beginning at the byte named in INDEX.

EXAMPLE:

```
PROCEDURE DELETE_DEMO;
VAR
  LONG_STR : STRING;
BEGIN
  LONG_STR := '    get rid of the leading blanks';
  WRITELN(LONG_STR);
  DELETE(LONG_STR,1,POS('g',LONG_STR)-1);
  WRITELN(LONG_STR);
END;
```

Output:

```
      get rid of the leading blanks
get rid of the leading blanks
```

5.15 INSERT

```
PROCEDURE INSERT( SOURCE, DESTINATION, INDEX);
```

DESTINATION is a string. SOURCE is a character or string, literal or variable. INDEX is an integer expression. Insert the SOURCE into the DESTINATION at the location specified in INDEX.

EXAMPLE:

```
PROCEDURE INSERT_DEMO;
VAR
  LONG_STR : STRING;
  S1 : STRING[10];
BEGIN
  LONG_STR := 'Remember  Luke';
  S1 := 'the Force,';
  INSERT(S1,LONG_STR,10);
  WRITELN(LONG_STR);
  INSERT('to use ',LONG_STR,10);
  WRITELN(LONG_STR);
end;
```

```
Output:
Remember the Force, Luke
Remember to use the Force, Luke
```

5.16 ASSIGN

```
PROCEDURE ASSIGN( FILE, NAME );
```

This procedure is used to assign an external file name to a file variable prior to a RESET or a REWRITE. FILE is a file name, NAME is a literal or a variable string containing the name of the file to be created. FILE must be of type TEXT to use the special device names below.

The user should note that standard Pascal defines a "local" file. Pascal/MT+ implements this facility using temporary file names in the form PASTMPxx.$$$ where xx is sequentially assigned starting at zero at the beginning of each program. If an external file REWRITE is not preceeded by an ASSIGN then a temporary file name will also be assigned to this file before creation.

NAME is normally a CP/M disk file name in the standard format: d:filename.ext but can also be a special device name:

Device names

CON: - When used as input will echo input characters and echo CR as CR/LF and backspace [CHR(8)] as backspace, space, backspace

When used as output will echo CR as CR/LF and CP/M will expand tabs to every 8 character positions.

KBD: - CP/M console, input device only. No echo or interpretation

TRM: - CP/M console, output device only. No interpretation

LST: - CP/M printer, output device only. No interpretation including no tab expansion

Examples of ASSIGN usage:

```
ASSIGN(F,'A:MT280.OVL');
ASSIGN(CONIN,'CON:');
ASSIGN(KEYBOARD,'KBD:');
ASSIGN(CRT,'TRM:');
ASSIGN(PRINTFILE,'LST:');
```

5.17 WNB, GNB

```
FUNCTION GNB(FILEVAR: FILE OF PAOC):CHAR;
FUNCTION WNB(FILEVAR: FILE OF CHAR; CH:CHAR) : BOOLEAN;
```

These functions allow the user to have BYTE level access to a file in a high speed manner. PAOC is any type which is fundamentally a Packed Array Of Char. The size of the packed array is optimally in the range 128..4095.

GNB will allow the user to read a file a byte-at-a-time. It is a function which returns a value of type CHAR. The EOF function will be valid when the physical end-of-file is reached but not based upon any data in the file (such as Ctrl/Z in CP/M TEXT files).

WNB will allow the user to write a file a byte-at-a-time. It is a function which requires a file and a character to write. It returns a boolean value which is true if there was an error while writing that byte to the file. No interpretation is done on the bytes which are written.

The reason GNB and WNB are used (as opposed to F^, GET/PUT combinations) is that they are significantly faster.

5.18 BLOCKREAD, BLOCKWRITE

```
BLOCKREAD (F:FILEVAR; BUF:ANY; VAR IOR:INTEGER; SZ,RB:INTEGER);
BLOCKWRITE(F:FILEVAR; BUF:ANY; VAR IOR:INTEGER; SZ,RB:INTEGER);
```

These procedures are used for direct CP/M disk access. FILEVAR is an untyped file (FILE;). BUF is any variable which is large enough to hold the data. IOR is an integer which receives the returned value from the CP/M BDOS. SZ is the number of bytes to transfer and RB is the relative block number.

The data is transfered either to or from the users BUF variable for the specified number of bytes.

For CP/M environments the SZ must be an multiple of 128 and RB must be in the range Ø..127.

5.19 OPEN, OPENX

```
PROCEDURE OPEN  ( FILE, TITLE, RESULT );
PROCEDURE OPENX ( FILE, TITLE, RESULT, EXTENT );
```

The OPEN and OPENX procedures are provided to increase the flexibility of Pascal/MT+ and to provide compatibility with previous releases of Pascal/MT. FILE is any file type variable. TITLE is a string. RESULT is a VAR INTEGER parameter. EXTENT is an INTEGER expression.

The OPEN procedure is exactly the same as executing an ASSIGN(FILE,TITLE), RESET(FILE) and RESULT := IORESULT sequence.

The OPENX procedure will set the extent number in the file control block before opening the file to support CP/M file extent manipulation.

EXAMPLES:

```
OPEN ( INFILE, 'A:FNAME.DAT', RESULT );

OPENX( INFILE, 'C:TESTNAME.FIL', RESULT, RECNUM DIV 128);
```

5.20 CLOSE, CLOSEDEL

```
PROCEDURE CLOSE    ( FILE, RESULT );
PROCEDURE CLOSEDEL ( FILE, RESULT );
```

The CLOSE and CLOSEDEL procedures are used for closing and closing with delete respectively. The CLOSE procedure must be called to guarantee that data written to a file using any method is properly purged from the file buffer to the disk. The CLOSEDEL is normally used on temporary files to delete them after use. FILE and RESULT are the same as used in OPEN (see section 5.19).

Files are implicitly closed when an open file is RESET or at the normal end of program execution. No more than 10 simultaneously open files will be automatically closed. The user may have more than 10 files open simultaneously but only the first ten files opened will be automatically closed.

5.21 PURGE

```
PROCEDURE PURGE( FILE );
```

The PURGE procedure is used to delete a file whose name is stored in a string. The user must first ASSIGN the name to the file and then execute PURGE. Note: in a CP/M environment there is no return value from CP/M on file deletions and the IORESULT will always be Ø after a PURGE.

EXAMPLE:

```
ASSIGN(F,'B:BADFILE.BAD');

PURGE(F);                          (* DELETE B:BADFILE.BAD *)
```

5.22 IORESULT

```
FUNCTION IORESULT : INTEGER;
```

After each I/O operation the value which is returned by the IORESULT function is set by the run-time library routines. In general the value of IORESULT is system dependent and on CP/M reflects the result of the returned value from the BDOS. In a CP/M environment the general rule is that 255 means an error and any other value is an good result. This is not the case in CLOSE and WRITE/PUT/WNB. In these procedures a non-zero IORESULT value means error.

EXAMPLE:

```
ASSIGN(F,'C:HELLO');
RESET(F);

IF IORESULT = 255 THEN
  WRITELN('C:HELLO IS NOT PRESENT');
```

Listed below are IORESULT values for CP/M:

PROCEDURE	VALUES
CLOSE	255 MEANS ERROR, ANYTHING ELSE IS OK
RESET	255 MEANS ERROR, ANYTHING ELSE IS OK
REWRITE	255 MEANS ERROR, ANYTHING ELSE IS OK
READ/READLN/GET	<> Ø MEANS END OF FILE, Ø MEANS OK
PAGE/WRITE/WRITELN/PUT	<> Ø MEANS ERROR, Ø MEANS OK

5.23 MEMAVAIL, MAXAVAIL

```
FUNCTION  MEMAVAIL : INTEGER;
FUNCTION  MAXAVAIL : INTEGER;
```

The functions MEMAVAIL and MAXAVAIL are used in conjunction with NEW and DISPOSE to manage the HEAP memory area in Pascal/MT+. The MEMAVAIL function returns the largest total available memory at any given time irrespective of fragmentation. The MAXAVAIL function will first garbage collect and then report the largest block available. The MAXAVAIL function can be used to force a garbage collect before a time sensitive section of programming.

The Pascal/MT+ system supports fully the NEW and DISPOSE mechanism defined by the Pascal Standard. In the CP/M environment the HEAP area grows from the end of the data area and the stack frame (for recursion) grows from the top of memory down. The hardware stack register in a CP/M environment is pre-loaded with the contents of absolute location 0006 unless the $Z toggle is used to override this. The stack frame grows starting at 512 bytes below the initialized hardware value. The user should refer to section 2.5 of the applications guide for more information on the $Z toggle.

5.24 Quick reference guide to built-ins

In alphabetical order within each group:

Character array manipulation routines:

```
PROCEDURE FILLCHAR ( DESTINATION, LENGTH, CHARACTER);
PROCEDURE MOVELEFT ( SOURCE, DESTINATION, NUM_BYTES);
PROCEDURE MOVERIGHT( SOURCE, DESTINATION, NUM_BYTES);
```

Bit and byte manipulation routines:

```
PROCEDURE CLRBIT( BASIC_VAR, BIT_NUM);
FUNCTION  HI    ( BASIC_VAR )          : INTEGER;
FUNCTION  LO    ( BASIC_VAR )          : INTEGER;
PROCEDURE SETBIT( BASIC_VAR, BIT_NUM);
FUNCTION  SHL   ( BASIC_VAR, NUM)      : INTEGER;
FUNCTION  SHR   ( BASIC_VAR, NUM)      : INTEGER;
FUNCTION  SWAP  ( BASIC_VAR )          : INTEGER;
FUNCTION  TSTBIT( BASIC_VAR, BIT_NUM)  : BOOLEAN;
```

String handling routines:

```
FUNCTION  CONCAT ( SOURCE1, SOURCE2,...,SOURCEn ) : STRING;
FUNCTION  COPY   ( SOURCE, LOCATION, NUM_BYTES)   : STRING;
PROCEDURE DELETE ( TARGET, INDEX, SIZE );
PROCEDURE INSERT ( SOURCE, DESTINATION, INDEX);
FUNCTION  LENGTH ( STRING )                       : INTEGER
FUNCTION  POS    ( PATTERN, SOURCE)               : INTEGER
```

File handling routines:

```
PROCEDURE ASSIGN    ( FILE, NAME );
PROCEDURE BLOCKREAD ( FILE, BUF, IOR, NUMBYTES, RELBLK);
PROCEDURE BLOCKWRITE( FILE, BUF, IOR, NUMBYTES, RELBLN);
PROCEDURE CLOSE     ( FILE, RESULT );
PROCEDURE CLOSEDEL  ( FILE, RESULT );
FUNCTION  GNB       ( FILE )       : CHAR
PROCEDURE IORESULT                 : INTEGER;
PROCEDURE OPEN      ( FILE, TITLE, RESULT );
PROCEDURE OPENX     ( FILE, TITLE, RESULT, EXTENT );
PROCEDURE PURGE     ( FILE );
FUNCTION  WNB       ( FILE, CHAR ) : BOOLEAN;
```

Miscellaneous routines:

```
FUNCTION  ADDR  ( VARIABLE REFERENCE )    : INTEGER;
PROCEDURE EXIT;
FUNCTION  MAXAVAIL : INTEGER;
FUNCTION  MEMAVAIL : INTEGER;
FUNCTION  SIZEOF( VARIABLE OR TYPE NAME) : INTEGER;
PROCEDURE WAIT  ( PORTNUM, MASK, POLARITY);
```

6.0 Interrupt procedures

A special procedure type is implemented in Pascal/MT+: the interrupt procedure. The user selects the vector to be associated with each interrupt. The procedure is declared as follows:

PROCEDURE INTERRUPT [<vec num>] <identifier> ;

Interrupt procedures may not have parameters lists but may have local variables and access global variables. The compiler generates code at the begining of the program to load the vector with the procedure address. For 8080/Z80 systems the vector number should be in the range of 0..7. For other systems consult the applications guide for the appropriate processor. For Z80 mode 2 interrupts the user may declare an ABSOULUTE variable to allocate an interupt table and then use the ADDR function to fill in this table. The INLINE facility woule be used in a Z80 environment to initialize the I-register.

The compiler generates code to push the registers at the begining of procedure execution and pop the registers and re-enable interrupts at the end of execution of the procedure. The compiler implements two built-in procedures ENABLE and DISABLE to control the hardware interrupt flag.

The user should note that the system does not generate re-entrant code. Typically interrupt procedures will set global variables and not perform other procedure calls or input / output. CP/M users should note that I/O through the CP/M BDOS typically re-enables interrupts.

Listed below is a simple example program which waits for one of four switches to interrupt and then toggles the state of a light which is attached to the switch. The I/O ports for the lights are 0..3 and the switches interrupt using restarts 2, 3, 4 and 5.

```
PROGRAM INT_DEMO;

CONST
  LIGHT1 = 0;           (* DEFINE I/O PORT CONSTANTS *)
  LIGHT2 = 1;
  LIGHT3 = 2;
  LIGHT4 = 3;

  SWITCH1 = 2;          (* DEFINE INTERRUPT VECTORS *)
  SWITCH2 = 3;
  SWITCH3 = 4;
  SWITCH4 = 5;

VAR
  LIGHT_STATE : ARRAY [LIGHT1..LIGHT4] OF BOOLEAN;
  SWITCH_PUSH : ARRAY [LIGHT1..LIGHT4] OF BOOLEAN;

  I : LIGHT1 .. LIGHT4;

PROCEDURE INTERRUPT [ SWITCH1 ] INT1;
BEGIN
  SWITCH_PUSH[LIGHT1] := TRUE
END;

PROCEDURE INTERRUPT [ SWITCH2 ] INT2;
BEGIN
  SWITCH_PUSH[LIGHT2] := TRUE
END;

PROCEDURE INTERRUPT [ SWITCH3 ] INT3;
BEGIN
  SWITCH_PUSH[LIGHT3] := TRUE
END;

PROCEDURE INTERRUPT [ SWITCH4 ] INT4;
BEGIN
  SWITCH_PUSH[LIGHT4] := TRUE
END;


BEGIN (* MAIN PROGRAM *)

  (* INITIALIZE BOTH ARRAYS *)

  FOR I := LIGHT1 TO LIGHT4 DO
    BEGIN
      LIGHT_STATE[I] := FALSE; (* ALL LIGHTS OFF *)
      SWITCH_PUSH[I] := FALSE;  (* NO INTERRUPTS YET *)
    END;

  ENABLE;     (* LET THE USERS HAVE AT IT! *)

  REPEAT
```

```
    REPEAT      (* UNTIL INTERRUPT *)
    UNTIL SWITCH_PUSH[LIGHT1] OR SWITCH_PUSH[LIGHT2] OR
          SWITCH_PUSH[LIGHT3] OF SWITCH_PUSH[LIGHT4];

    FOR I := LIGHT1 TO LIGHT4 DO (* SWITCH LIGHTS *)
      IF SWITCH_PUSH[I] THEN
        BEGIN
          SWITCH_PUSH[I] := FALSE;
          LIGHT_STATE[I] := NOT LIGHT_STATE[I]; (* TOGGLE IT *)
          OUT[I] := LIGHT_STATE[I]
        END

  UNTIL FALSE; (* FOREVER DO THIS LOOP *)

END. (* OF PROGRAM *)
```

7.0 INLINE AND Mini assembler

Pascal/MT+ has a very useful built-in feature called INLINE. This feature allows the user to insert data in the middle of a Pascal/MT+ procedure or function. In this way small machine code sequences and constant tables may be inserted into a Pascal/MT+ program without using externally assembled routines.

7.1 Syntax

The syntax for the INLINE feature is very similar to that of a procedure call in Pascal. The word INLINE is used followed by a left parenthesis '(' followed by any number of arguments separated by the slash '/' character and terminated by a right parenthesis ')'. The arguments between the slashes must be constants or variable references which evaluate to constants. These constants can be of any of the following types: CHAR, STRING, BOOLEAN, INTEGER or REAL. The user should note that a STRING in quotes does not generate a length byte but simply the data for the string. Note that in stack frame addressing (either by using $S+ or on more sophisticated CPUs) variables will evaluate to the offset into the appropriate data segment. For CPUs which use static addresssing (e.g. 8080, 8085 and Z80) the address is the absolute address of the data.

Literal constants which are of type integer will be allocated one byte if the value falls in the range 0..255. This is not the case for named, declared, integer constants which will always be allocated two bytes.

In addition to constant data the Pascal/MT+ system also provides a built-in Mini assembler feature for 8080/8085 CPUs. The user may place the assembly language mnemnonic after a double quote and the first phase of the compiler will translate this mnemonic into the appropriate hex value (e.g. "MOV A,M will translate into $7E). In the future this may be extended to handle other processors.

EXAMPLE

```
INLINE( "LHLD  /        (* LHLD OPCODE FOR 8080 *)
        VAR1   /        (* REFERENCE VARIABLE    *)
        "SHLD  /        (* SHLD OPCODE FOR 8080 *)
        VAR2   );       (* REFERENCE VARIABLE    *)
```

7.2 Applications

The INLINE facility can be used to insert native machine code or to build compile-time tables. The following two sections give examples of each of these uses.

7.2.1 Code examples

The code below gives an example of how to use the INLINE facility to write a procedure which calls CP/M and returns a value. This routine is present in the run-time library as @BDOS.

EXAMPLE:

```
FUNCTION @BDOS(FUNC:INTEGER; PARM:WORD):INTEGER;
CONST
  CPMENTRYPOINT = 5;    (* SO IT ALLOCATES 2 BYTES *)
VAR
  RESULT : INTEGER;     (* SO WE CAN STORE IT HERE *)

BEGIN
  INLINE( $2A / FUNC /           (* LHLD FUNC   *)
          $4D /                  (* MOV C,L     *)
          $2A / PARM /           (* LHLD PARM   *)
          $EB /                  (* XCHG        *)
          $CD / CPMENTRYPOINT / (* CALL BDOS   *)
          $6F /                  (* MOV L,A     *)
          $26 / $00 /            (* MVI H,0     *)
          $22 / RESULT );        (* SHLD RESULT *)

  @BDOS := RESULT;      (* SET FUNCTION VALUE *)
END;
```

7.2.2 Constant data generation

The program fragment below demonstrates how the INLINE facility can be used to construct a compile time table:

```
EXAMPLE:

PROGRAM DEMO_INLINE;

TYPE
    IDFIELD = ARRAY [1..4] OF ARRAY [1..10] OF CHAR;

VAR
  TPTR : ^IDFIELD;

PROCEDURE TABLE;
BEGIN
  INLINE(     'MTMICROSYS' /
              'SOFTWARE  ' /
              'POWER     ' /
              'TOOLS.....' );
END;

BEGIN (* MAIN PROGRAM *)
  TPTR := ADDR(TABLE);

  WRITELN(TPTR^[3]);              (* SHOULD WRITE 'POWER     ' *)

END.
```

8.0 INP and OUT arrays

The Pascal/MT+ system provides a feature which allows direct manipulation of Input and Output hardware ports. Two pre-declared arrays, INP and OUT, are provided which are of type BYTE and may be subscripted with port number constants and expressions. The INP array may be used only in expressions and the OUT array may be used only on the LEFT hand side of an assignment statement. For those processors which have WORD Input and Output ports two additional arrays INPW and OUTW are also declared.

The following discussion is specific to the 8080 and Z80 type CPUs. The arrays may be subscripted with integer expressions in the range 0..255. If constant subscripts are used the code is generated in-line. If expressions are used a call is made to the appropriate run-time library routines to handle variable port I/O. If the values from INP are assigned to variables of type INTEGER the most significant byte will contain 00.

For use with the 8085 the Pascal/MT+ system supports the built-in names RIM85 and SIM85 which allow direct manipulation of the RIM and SIM ports on the 8085 CPU. RIM85 may be used to subscript the INP array and SIM85 may be used to subscript the OUT array.

9.Ø Chaining

There are times when programs exceed the memory available and also many times when segmentation of programs for compilation and maintenance purposes is desired. The Pascal/MT+ system provides a "chaining" mechanism in which one program may transfer control to another program.

The user must declare an untyped file (FILE;) and use the ASSIGN and RESET procedures to initialize the file. Following this the user may execute a call to the CHAIN procedure passing the name of the file variable as a single parameter. The run-time library routine will then perform the appropriate functions to load in the file opened by the user using the RESET statement. The size of the various programs does not matter. This means that a small program may chain to a large one and a large program may chain to a small one. If the user desires to communicate between the chained program the user may choose to communicate in two ways: shared global variables and ABSOLUTE variables.

Using the shared global variable method the user must guarantee that at least the first section of global variables be exactly the same in the two programs that wish to communicate. The remainder of the global variables need not be the same and the declaration of external variables in the global section will not affect this mapping. In addition to having matching declarations the user must use the /D option switch available in the linker (see section 3 of the applications guide) to place the variables at the same location in all programs that wish to communicate.

Using the ABSOLUTE variable method the user would typically define a record which is used as a communication area and then define this record at an absolute location in each module. This does not require the use of the /D switch in the linker but does require knowledge of the memory used by the program and the system.

Listed below are two example programs which communicate with each other using the ABSOLUTE variable method and the first program will CHAIN to the second program which will print the results of the first program's execution:

EXAMPLE:

```
PROGRAM PROG1;

TYPE
  COMMAREA = RECORD
               I,J,K : INTEGER
             END;

VAR
  GLOBALS : ABSOLUTE [$8000] COMMAREA;
  CHAINFIL: FILE;

BEGIN (* MAIN PROGRAM #1 *)
  WITH GLOBALS DO
    BEGIN
      I := 3;
      J := 3;
      K := I * J
    END;

  ASSIGN(CHAINFIL,'A:PROG2.COM');
  RESET(CHAINFIL);
  IF IORESULT = 255 THEN
    BEGIN
      WRITELN('UNABLE TO OPEN PROG2.COM');
      EXIT
    END;

  CHAIN(CHAINFIL)
END.                    (* END PROG1 *)
```

```
(* PROGRAM #2 IN CHAIN DEMONSTRATION *)

PROGRAM PROG2;

TYPE
  COMMAREA = RECORD
               I,J,K : INTEGER
             END;

VAR
  GLOBALS : ABSOLUTE [$8000] COMMAREA;


BEGIN (* PROGRAM #2 *)
  WITH GLOBALS DO
    WRITELN('RESULT OF ',I,' TIMES ',J,' IS =', K)

END.                    (* RETURNS TO OPERATING SYSTEM WHEN COMPLETE *
```

10.0 Disassembler

The disassembler component of the Pascal/MT+ package combines the .PRN file produced by the first phase of the compiler with the .ERL file produced by the last phase of the compiler into a human readable file which contains assembly language coding interspersed with the Pascal/MT+ statements. This allows investigation into the code produced by the compiler and provides the necessary information when it is required to debug the object code at the machine code level.

The disassembler is a stand-alone program which is invoked by specifying the name of the disassembler, the name of the .PRN file, the name of the .ERL file and the name of the output file:

DIS???? <input name> {<destination name> {,L=nnn}}

10.1 Instructions

Here ???? is the type of CPU (e.g. 8080, Z80, 68K, 8086, etc.). The disassembler looks for a .ERL and a .PRN file with <input name> as a prefix. These files may be on any disk but both must be on the same disk. The destination file name may be a CP/M file name or a Pascal/MT+ device name such as CON: or LST:. The default destination name is CON:. The L=nnn parameter allows the user to specify the number of lines per page on the output device. This is useful when using printers such as the T.I. 810 which has a 6-lines-per-inch or 8-lines-per-inch switch. Using 8-lines-per-inch the user should specify (for 11" paper) that the paper has 88 lines. This can save considerable ammounts of paper. To use the L= option the user MUST specify the <destination name>.

10.2 Sample

The following Pascal/MT+ program was compiled and run through the disassembler and produced the following output (for an 8080/Z80):

Input program:

```
PROGRAM PPRIME;
CONST
  SIZE=8190;
VAR
  PRIME: ARRAY[0..SIZE] OF BOOLEAN;
  I,J,K,L: INTEGER;
  COUNT: INTEGER;
  CH : CHAR;
  MAX: 0..SIZE;

EXTERNAL PROCEDURE X1;
EXTERNAL PROCEDURE X2;
EXTERNAL PROCEDURE X3;

(*$P*)

PROCEDURE TEST1(A,B,C:INTEGER);
BEGIN
  B:=SUCC(SUCC(SUCC(A+A)));
  C:=A+B;
  WHILE C<=MAX DO
   BEGIN
     PRIME[C]:=FALSE;
     C:=C+B;
   END;
END; (* TEST1 *)

(*$P*)

BEGIN
  MAX := SIZE;
  WRITE('G');
  READ(CH);
  FOR L := 1 TO 10 DO
    BEGIN
      COUNT:=0;
      FILLCHAR(PRIME,SIZEOF(PRIME),CHR(TRUE));

      FOR I:=0 TO MAX DO
        IF PRIME[I] THEN
         BEGIN
           TEST1(I,J,K);
           COUNT:=SUCC(COUNT);
         END;
    END;
  WRITELN(COUNT);
```

```
  WRITE('E');
END.
```

Output from disassembler:

The user will note that references to program locations are followed by a single quote (1000') and references to data locations are followed by a double quote (0000").

The user will also note that the operand of instructions which reference external variables point to the previous reference and the final reference contains absolute 0000. The list of external chains is following the disassembly of the program.

```
Pascal/MT+  5.00  Copyright (c) 1980 by MT MicroSYSTEMS  Page #    1
Disassembly of: TESTIT


Stmt  Nest     Source Statement / Symbolic Object Code

       PRIME     EQU     0000
       L         EQU     2000
       K         EQU     2002
       J         EQU     2004
       I         EQU     2006
       COUNT     EQU     2008
       CH        EQU     200A
       MAX       EQU     200C
   1     0     PROGRAM PPRIME;

0000             DB      00,00,00,00,00,00,00,00
0008             DB      00,00,00,00,00,00,00,00
0010             JMP     0000
0013             JMP     0000

   2     0     CONST
   3     1       SIZE=8190;
   4     1     VAR
   5     1       PRIME: ARRAY[0..SIZE] OF BOOLEAN;
   6     1       I,J,K,L: INTEGER;
   7     1       COUNT: INTEGER;
   8     1       CH : CHAR;
   9     1       MAX: 0..SIZE;
  10     1
  11     1     EXTERNAL PROCEDURE X1;
  12     1     EXTERNAL PROCEDURE X2;
  13     1     EXTERNAL PROCEDURE X3;
  14     1
  15     1     (*$P*)
  16     1
  17     1     PROCEDURE TEST1(A,B,C:INTEGER);
  18     1     BEGIN
       TEST1:

0016             CALL    0000
```

```
0019            POP     H
001A            SHLD    200E"
001D            POP     H
001E            SHLD    2010"
0021            POP     H
0022            SHLD    2012"
0025            CALL    0000

  19    2       B:=SUCC(SUCC(SUCC(A+A)));

0028            LHLD    2012"
002B            XCHG
002C            LHLD    2012"
002F            DAD     D
0030            INX     H
0031            INX     H
0032            INX     H
0033            SHLD    2010"

  20    2       C:=A+B;

0036            LHLD    2012"
0039            XCHG
003A            LHLD    2010"
003D            DAD     D
003E            SHLD    200E"

  21    2       WHILE C<=MAX DO

0041            LHLD    200E"
0044            PUSH    H
0045            LHLD    200C"
0048            PUSH    H
0049            CALL    0000
004C            POP     PSW
004D            JNC     006D'

  22    2        BEGIN
  23    3          PRIME[C]:=FALSE;

0050            LXI     H,0000"
0053            XCHG
0054            LHLD    200E"
0057            DAD     D
0058            PUSH    H
0059            LXI     H,0000
005C            XCHG
005D            POP     H
005E            MOV     M,E

  24    3          C:=C+B;

005F            LHLD    200E"
0062            XCHG
0063            LHLD    2010"
```

```
0066              DAD     D
0067              SHLD    200E"

  25     3,       END;

006A              JMP     0041'

  26     2    END; (* TEST1 *)

006D              RET

  27     1.
  28     1    (*$P*)
  29     1
  30     1    BEGIN

006E              LHLD    0006
0071              SPHL
0072              CALL    0000

  31     1      MAX := SIZE;

0075              LXI     H,1FFE
0078              SHLD    200C"

  32     1      WRITE('G');

007B              LXI     H,0000
007E              PUSH    H
007F              CALL    0000
0082              LXI     H,0047
0085              PUSH    H
0086              CALL    0000
0089              CALL    0000

  33     1      READ(CH);

008C              LXI     H,200A"
008F              PUSH    H
0090              LXI     H,0000
0093              PUSH    H
0094              CALL    0080'
0097              CALL    0000

  34     1      FOR L := 1 TO 10 DO

009A              LXI     H,0001
009D              PUSH    H
009E              LXI     H,000A
00A1              PUSH    H
00A2              POP     D
00A3              POP     H
00A4              DCX     H
00A5              SHLD    2000"
00A8              INX     H
```

```
00A9            PUSH    H
00AA            PUSH    D
00AB            CALL    0000
00AE            SHLD    2014"
00B1            LHLD    2000"
00B4            INX     H
00B5            SHLD    2000"
00B8            LHLD    2014"
00BB            DCX     H
00BC            SHLD    2014"
00BF            MOV     A,H
00C0            ORA     L
00C1            JZ      012C'

  35    1       BEGIN
  36    2         COUNT:=0;

00C4            LXI     H,0000
00C7            SHLD    2008"

  37    2         FILLCHAR(PRIME,SIZEOF(PRIME),CHR(TRUE));

00CA            LXI     H,0000"
00CD            PUSH    H
00CE            LXI     H,1FFF
00D1            PUSH    H
00D2            LXI     H,0001
00D5            PUSH    H
00D6            CALL    0000

  38    2
  39    2         FOR I:=0 TO MAX DO

00D9            LXI     H,0000
00DC            PUSH    H
00DD            LHLD    200C"
00E0            PUSH    H
00E1            POP     D
00E2            POP     H
00E3            DCX     H
00E4            SHLD    2006"
00E7            INX     H
00E8            PUSH    H
00E9            PUSH    D
00EA            CALL    00AC'
00ED            SHLD    2016"
00F0            LHLD    2006"
00F3            INX     H
00F4            SHLD    2006"
00F7            LHLD    2016"
00FA            DCX     H
00FB            SHLD    2016"
00FE            MOV     A,H
00FF            ORA     L
0100            JZ      0129'
```

```
  40    2           IF PRIME[I] THEN

0103              LXI     H,0000"
0106              XCHG
0107              LHLD    2006"
010A              DAD     D
010B              MOV     A,M
010C              RAR
010D              JNC     0126'

  41    2           BEGIN
  42    3             TEST1(I,J,K);

0110              LHLD    2006"
0113              PUSH    H
0114              LHLD    2004"
0117              PUSH    H
0118              LHLD    2002"
011B              PUSH    H
011C              CALL    0013'

  43    3             COUNT:=SUCC(COUNT);

011F              LHLD    2008"
0122              INX     H
0123              SHLD    2008"

  44    3           END;

0126              JMP     00F0'

  45    2         END;

0129              JMP     00B1'

  46    1       WRITELN(COUNT);

012C              LHLD    2008"
012F              PUSH    H
0130              LXI     H,007C'
0133              PUSH    H
0134              CALL    0095'
0137              CALL    0087'
013A              CALL    0000
013D              CALL    0000

  47    1       WRITE('E');

0140              LXI     H,0131'
0143              PUSH    H
0144              CALL    0135'
0147              LXI     H,0045
014A              PUSH    H
014B              CALL    0138'
```

```
014E              CALL    008A'

  48      1     END.

0151              CALL    0000

External reference chain @WIN    --> 013B
External reference chain @CHW    --> 014F
External reference chain @RCH    --> 0098
External reference chain @PST    --> 0017
External reference chain @PLD    --> 0026
External reference chain @CRL    --> 013E
External reference chain @LEI    --> 004A
External reference chain @FIN    --> 00EB
External reference chain @SFB    --> 0145
External reference chain @DWD    --> 014C
External reference chain @INI    --> 0073
External reference chain @HLT    --> 0152
External reference chain OUTPUT  --> 0141
External reference chain INPUT   --> 0091
External reference chain FILLCH  --> 00D7
```

11.0 Debugger

The Pascal/MT+ debugger is a component of the Pascal/MT+ system which is linked into the object program along with the run-time support library (from DEBUGGER.ERL). The user must link the debugger as the first module of the program so that execution begins with the debugger when the program is run.

The compiler produces a .PSY file for each module when the D switch is specified to the MTPLUS program. These .PSY files contain records for each procedure, function and variable declared in the program. The address fields for each of these items is module relative. Link/MT+ will process these .PSY files and create a .SYP file containing absolute addresses for the procedures, functions and variables. The debugger then uses this .SYP file for symbolic variable display, symbolic breakpoints, etc.

The debugger can display variables, set breakpoints, single step a statement at a time, display symbol tables, and display entry and exit from procedures and functions.

The debugger can be used in a non-CP/M environment if the user responds with simply <return> to the debugger's request for the .SYP file name. This disables only the symbolic facilities but retains the display by address facilities.

The following two sections describe how to include the debugger code in an object program and how to operate the debugger.

11.1 Instructions

To include debugger information into the object program the user must specify the D switch to MTPLUS.COM. The compiler will then produce a .PSY file to the same disk as the .ERL file. In addition the compiler will generate code at the begining of each line and at the begining and end of of each procedure and function. The $D toggle controls the generation of this code. The default state of $D is on ($D+) when the D switch is specified to MTPLUS.COM. The user may turn the $D toggle off ($D-) around procedures and functions which have been debugged or are time critical. The $D toggle (as described in section 2.5 of the applications guide) may be switched on and off as desired around procedures and functions.

Link/MT+ (as described above) creates a .COM and a .SYP file from the .ERL and .PSY files created by the compiler.

The debugger will ask for the name of the symbol table file when executed. The user should respond with the name of the .SYP file or <return> for no symbols. The debugger will then respond with '+>'. The user may then enter any of the debugger commands and proceed to debug the program under test.

11.2 Commands

The debugger converts items whenever possible into the form expected by the user (i.e. decimal for integers, TRUE / FALSE for booleans, etc.). When this is not possible the debugger will display the data in HEX and ASCII. Listed below are the syntax elements and then the commands.

The term <name> is either a variable name, a procedure / function name, or a prefixed variable name. A prefixed name is a variable name prefixed with a procedure / function name. Names are 1 to 8 characters long and follow the syntax of the Pascal compiler. Underscores are allowed and ignored (e.g. A_B is exactly the same as AB). This syntax is used to display local variables and parameters. If two procedures each have a local procedure of the same name only the first procedure linked will be available for symbolic display.

The term <num> is either a decimal number or, if prefixed by a '$' character, a hexadecimal number. Decimal numbers fall in the range 0..32767. Hexadecimal numbers in the range 0..FFFF (for 64K machines, the range is larger for 8086/8088, Z8000 and 68000).

```
<name>          ::= <identifier> : <identifier> |
                    <identifier>                |
                    <num>

<num>           ::= $ <hex number> |
                    <decimal number>
```

Command Syntax	Meaning
DV <name> {^}	Display Variable - variable display by <name>. If this is a pointer var the contents of the pointer is displayed unless followed by ^ which causes the data pointed to by the pointer to be displayed. (e.g. DS STR).

The following commands are used when symbols are not available or when fields within records or array elements are to be displayed:

Each of these commands is followed by a parameter in the form:

<parm> ::= [<name> | <num>] {^} {[+ | -] <num>}

Examples:

```
(* Pascal declarations: *)

TYPE
  PAOC = ARRAY [1..40] OF CHAR;

VAR
    ABC : INTEGER;
    PTR : ^PAOC;
```

Example of <parm>:

ABC	an integer
PTR^	entire array
ABC+10	arbitrary location
PTR^+10	PTR^[11]
ABC-3	arbitrary location
PTR^-3	arbitrary location
$3FFD	
$423B^	32 bytes pointed to by 423B
$3FFD+$5B	32 bytes at 4058
$423B^+49	32 bytes pointed to by contents of 423B + 49
PROC1:I	local variable
PROC2:J^+9	offset from local pointer

DI <parm>	Display Integer
DC <parm>	Display Character
DL <parm>	Display Logical (Boolean)
DR <parm>	Display Real
DB <parm>	Display Byte
DW <parm>	Display Word
DS <parm>	Display String
DX <parm>	Display eXtended (structures) This is always displayed in HEX / ASCII format

The following commands allow control of the user program:

TR	Trace - Execute one line and return
T<num>	Trace <num> lines and return
BE	BEgin execution (start program from beginning)
GO	Continue execution from a breakpoint
SB <name>	Set breakpoint at beginning of procedure <name>
RB <name>	Remove breakpoint at procedure <name>
E+	Enable display entry and exit of each procedure or function during execution
E-	Disable entry / exit display
PN	Display procedure names from .SYP file
VN <name>	Display all variables associated with procedure <name>

12.0 Run-time Environment

The code generated by the Pascal/MT+ compiler is true, native machine code. Run-time library routines are required on each processor to support files and any other features which are not supported by the native hardware but are required to implement the entire Pascal language. The following information is specific to the 8080/Z80, CP/M implementation of Pascal/MT+. The reader is referred to the applications notes for other CPUs.

The Pascal/MT+ compiler generates program modules which have a very simple structure. At the beginning of the module is located a jump table containing a jump to each procedure or function in the module. Space is reserved at the beginning of the jump table for the main program and this jump is unused if the module is a MODULE and not a PROGRAM. In addition, in a PROGRAM there are 16-bytes of header information (in the 8080/Z80 version these are NOPs) which may be used in future versions for hardware dependent initialization. At the beginning of the main program the compiler generates code to load the stack pointer based upon the contents of location 6 (+4200 if necessary) which is the CP/M standard. ROM based users will typically wish to place some INLINE code there to re-initialize the SP. Also the compiler generates a call to the @INI routine which initializes the INPUT and OUTPUT text files and the stack frame pointer used when the $S+ toggle is activated. Again ROM based users will typically wish to re-write the @INI routine to suit their needs.

The Pascal/MT+ system requires subroutines from the run-time library in order to support the whole of the Pascal language. Some processors require less run-time support than others but in general all I/O is done via library routines and SET variables are manipulated via library routines. Only the run-time routines needed for a particular program are actually loaded when the program is linked with Link/MT+.

Included in this section is also a discussion of how to adapt the run-time routines for non-CP/M operation as is required for ROM based systems.

12.1 Library routines

Listed below are the names of all the run-time library routines and their function. For a description of their parameters and more detailed information the user is refered to the source code which accompanies this software package.

ROUTINE	FUNCTION
@CHN	Program chaining routine
@MUL	Integer multiply 16-bit stack
@MUX	Integer multiply 16-bit register
@FIN	FOR loop initialization helper.
@EQD @NED @GTD @LTD @GED @LED	. . . . String comparison routines for =, <>, >, <, >=, and <=
@EQS	Set equality
@NES	Set in-equality
@GES	Set superset
@LES	Set subset
@HLT	End of program halt routine, return to CP/M
@PST	Store ret addr temporarily
@PLD	Return ret addr to stack
@SAD	Set union
@SSB	Set difference
@SML	Set intersection
@SIN	Set membership
@BST	Build singleton set
@BSR	Build subrange set
@DYN	Load/Store in stack frame mode routine
@LNK	Allocate stack variable space
@ULK	De-allocate stack variable space
@EQA @NEA @GTA @LTA @GEA @LEA	 Array comparison routines =, <>, >, <, >= and <=
@XJP	Table Case Jump routine
@LBA	Load concat string buffer address
@ISB	Init string buffer
@CNC	Concatenate a string to the buffer
@CCH	Concatenate a char to the buffer
@RCH	Read a char from a file
@CRL	Write a newline (CR) to a file

@CWT	Wait for EOLN to be true on a file
@INP	Handle variable port input
@OUT	Handle variable port output
@WIN	Write an integer to a file
@RST	Read a string from a file
TSTBIT	Test for a bit on
SETBIT	Turn a bit on
CLRBIT	Turn a bit off
SHL	Shift a word left
SHR	Shift a word right
@EQI @NEI @GTI @LTI @GEI @LEI	Integer comparisons
@EQB @NEB @GTB @LTB @GEB @LEB	Boolean comparisons
@SFB	Set global FIB address
@DWD	Set default width and decimal places
@SIA	Reset input vector
@SOA	Reset output vector
@DIO	Set I/O vectors to default addresses
@INI	Run-time initialization
@STR	String store
@GETCH	Read a char from a file onto stack
@WCH	Write a string to a file
@DIV	16-bit DIV software routine
@MOD	16-bit MOD software routine
@XDIVD	utility divide routine used by @WIN
@MVL MOVE MOVELE	Block move left end to left end stack parms
@MVR MOVERI	Block move right end to right end stack parms
@PUTCH	Write a char from stack

@LEAD	Handle width in char outputs
@CHW	Write a char to a file
@CHW1	entry point used by @WCH and others
@EQR	
@NER	
@GTR	
@LTR	
@GER	Real comparisons
@LER	=, <>, >, <, >=, and <=
@RRL	Read a real from a file
@WRL	Write a real to a file
@RAD	Real add
@RSB	Real subtract
@RML	Real multiply
@RDV	Real divide
@RNG	Real negate
@RAB	Real absolute value
@XOP	Real utility load/store routine
SQRT	Real square root
TRUNC	
ROUND	Pascal built-in functions
IOERR	Used for unimplement I/O routines
CHAIN	Pascal interface for @CHN
OPEN	
OPENX	
BLOCKR	
BLOCKW	
CREATE	
CLOSE	
CLOSED	
GNB	
WNB	
PAGE	
EOLN	
EOF	
RESET	
REWRIT	
GET	
PUT	
ASSIGN	
PURGE	
IORESU	Run time support for files
COPY	
INSERT	
DELETE	

POS	Run time support for strings
@WNC	Write next char to a file
@RNC	Read next char from a file
@RIN	Read integer from a file
@S2I	Convert string to integer
@RNB	Read n bytes from a file
@WNB	Write n bytes to a file
@BDOS	Call CP/M directly
@SPN	Check for device names
@NOK	Check for legal file names
@NEW	Allocate memory for NEW procedure
@DSP	Deallocate memory for DISPOSE procedure
MEMAVA	MEMAVAIL function
MAXAVA	MAXAVAIL function

12.2 Console I/O

In Pascal/MT+ all I/O is file I/O and is vectored through the @SYSIN and @SYSOUT vectors which are located in and initialized by the @INI routine to point to the @RNC (read-next-char) routine for input and @WNC routine (write-next-char) for output. When re-directed I/O is used the @SIA and @SOA routines are used to change these vectors and the @DIO routine is used to reset these vectors at the end of a re-directed I/O statement.

In environments where minimum space is a concern and no file I/O is being used the user may simply rewrite the @RNC and @WNC routines and provide total console I/O support. Note that these routines must manipulate the INPUT FIB FEOF and FEOLN boolean variables if EOF, EOLN and READLN are to operate properly.

In the CP/M environment on 8080 and Z80 machines the @RNC and @WNC routines call GET and PUT which call @RNB and @WNB which call @BDOS and therefore cause about 2K bytes of software to be loaded even for a program which does console I/O only.

Users which need to operate in ROM environments should see section 12.4.

12.3 File I/O

In Pascal/MT+ all the file I/O routines (with the exception of the conversion routines) are written in Pascal and supplied in source code form. The reader will note that when looking at these routines one will see a definition of a data structure called a FIB or file-information-block. This FIB contains information about the current state of the file, a sector buffer, an FCB and other information. The organization of the FIB is known to all the Pascal routines and to some of the assembly language routines and should not be changed lightly.

In addition the reader will note that some of the routines have more parameters than normally found for those routines (such as RESET). The compiler recognizes when these built-in routines are being called and passes the buffer size along with the FIB address when calling these routines. Also the RESET routine is extra special in that the buffer size is passed as -1 if the file is a TEXT file so that interpretation of special characters and EOF can be handled properly.

12.4 ROM environments

The user may wish to run programs written in Pascal/MT+ in a ROM based system. This has been a design goal from the beginning and has been done successfully by many users. In order to perform formatted I/O in a ROM based environment the user must either use re-directed I/O for all READ and WRITE statements or rewrite the @RNC and @WNC routines mentioned in section 12.2. In addition the user of a ROM based system may wish to shorten and/or eliminate the INPUT and OUTPUT FIB storage located in the @INI module. This storage is required for TEXT file I/O compatibility but may not be needed in a ROM based environment. The user should be cautious and make sure that any changes to INPUT and OUTPUT are also handled correspondingly in @RST and @CWT.

Listed below are three skeletons for the @INI, @RNC and @WNC routines which can be used in ROM environments. The user should study the source code included with the package for additional details such as HEAP usage in ROM, etc.

```
;-----------------------------------------------------------------;
;       SAMPLE INITIALIZATION ROUTINE                             ;
;-----------------------------------------------------------------;

        PUBLIC  @INI
        PUBLIC  @SYSIN          ;SYSTEM INPUT VECTOR
        PUBLIC  @SYSOUT         ;SYSTEM OUTPUT VECTOR
        PUBLIC  INPUT           ;DEFAULT INPUT FIB
                                ;THIS MUST BE PRESENT EVEN IF NO
                                ;FILE I/O IS DONE
        PUBLIC  OUTPUT          ;AGAIN MUST BE PRESENT EVEN IF NO
                                ;FILE I/O IS DONE
        EXTRN   @RNC
        EXTRN   @WNC

@INI:
        LXI     H,@RNC
        SHLD    @SYSIN

        LXI     H,@WNC
        SHLD    @SYSOUT

;
;       ... ADD MORE HERE FOR HEAP, ETC. PRUNE FROM STANDARD @INI
;

        DSEG

@SYSIN: DS      2
@SYSOUT: DS      2

INPUT:  DS      1               ;DUMMY FIB
OUTPUT: DS      1               ;DUMMY FIB

        RET                     ;AND THAT'S A SIMPLE ONE
        END
```

```
;-----------------------------------------------------------------;
;       SAMPLE @RNC - READ NEXT CHARACTER ROUTINE                 ;
;-----------------------------------------------------------------;

        PUBLIC  @RNC
@RNC:
        ; INCLUDE CODE HERE TO GET CHARACTER INTO A-REG AND
        ; ECHO IT. ALSO IF USER WANTS TO SIMULATE CON: THE
        ; THE DRIVER MUST ECHO BACKSPACE AS <BACKSPACE, SPACE,
        ; BACKSPACE> AND CR AS CR/LF

        MOV     L,A
        MVI     H,Ø
        XTHL                        ;PUT FUNC VALUE ON STACK AND
                                    ;RET ADDR IN HL
        PCHL                        ;RETURN
        END
```

```
;-----------------------------------------------------------------;
;       SAMPLE @WNC - WRITE NEXT CHARACTER ROUTINE                ;
;-----------------------------------------------------------------;

        PUBLIC  @WNC

@WNC:
        POP     H               ;GET RET ADDR
        XTHL                    ;PUT IT BACK AND GET PARM CHAR

        ; CODE HERE TO WRITE CHARACTER IN L-REG TO OUTPUT DEVICE
        ; IF USER WANTS TO SIMULATE CON: COMPLETELY THE USER
        ; MUST OUTPUT CR AS CR/LF

        RET
        END
```

13.0 Pascal/MT+ : Assembly Interfacing

This section of the applications guide is intended to provide information for those Pascal/MT+ customers who wish to write and call assembly language routines from a Pascal/MT+ program. Included is a list of assemblers, required naming conventions, variable accessing, parameter passing conventions and restrictions on what assembly language features can be linked with LINK/MT+.

13.1 Assemblers

The assemblers used with Pascal/MT+ must generate the same relocatable format as the compiler. The 8080 and Z80 versions of the Pascal/MT+ system generate Microsoft compatible relocatable files. This is a bit stream relocatable format and is described in section 3 of this applications guide. This format is generated by the Microsoft M80 and the Digital Research RMAC assemblers. Both of these assemblers have been used successfully by MT MicroSYSTEMS to generate the run-time library.

13.2 Naming Considerations

The assemblers and the Pascal/MT+ compiler each generate entry point and external reference records in the relocatable file format. These records contain external symbol names. The Microsoft format allows for up to 7 character names but most assemblers only generate 6 character names and the Pascal/MT+ compiler will use all 7 characters. This means that if a variable is to be located in a Pascal/MT+ program and accessable to an assembly language routine by name, the user should limit the name to 6 characters.

In addition, M80 allows symbols to begin with $ and RMAC allows symbols to begin with ? neither of which is a legal identifier character in Pascal/MT+. M80 also does not consider $ to be a non-significant character but RMAC does. This means that in M80 the symbol A$B is actually placed in the relocatable file as A$B but in RMAC the same symbol would be in the file as AB. When using RMAC the use of $ to simulate the underscore (_) is often used but not transportable to M80.

13.3 Variable accessing

Accessing assembly language variables from Pascal and Pascal variables from assembly language is very simple.

To access assembly language variables from Pascal the variables should be declared as PUBLIC in the assembly language module and as EXTERNAL in the Pascal/MT+ program:

EXAMPLE:

```
; ASSEMBLY LANGUAGE PROGRAM FRAGMENT

        PUBLIC    XYZ

        DSEG


XYZ     DS        32            ;ACCESSABLE BY PASCAL

          .
          .
          .
        END
```

```
(* PASCAL PROGRAM FRAGMENT *)

VAR
   XYZ : EXTERNAL PACKED ARRAY [1..32] OF CHAR;
```

To access Pascal/MT+ GLOBAL variables from an assembly language program the user must declare the name to be EXTRN in the assembly language program and simply as a global variable (make sure the $E+ toggle is on!):

EXAMPLE:

```
; ASSEMBLY LANGUAGE PROGRAM FRAGMENT

        EXTRN     PQR


        LXI       H,PQR     ;GET ADDR OF PASCAL VARIABLE
          .
          .
          .
        END
```

```
(* PASCAL PROGRAM FRAGMENT *)

VAR (* IN GLOBALS *)
  PQR : INTEGER;            (* ACCESSABLE BY ASM ROUTINE *)
```

In addition to accessing the variables by name the user must know how the variables are allocated in memory. Section 4 of this applications guide discusses the storage allocation and format of each built-in scalar data type. Variables allocated in the GLOBAL data area are allocated essentially in the order shown. The exception being that variables which are in an identifier list before a type (e.g. A,B,C : INTEGER) are allocated in reverse order (i.e. C first, followed by B, followed by A). In some CPUs (such as the Z8000 and 68000) each variable declared on a separate line is allocated on a EVEN memory address boundary. Variables allocated on the same line which are not an even number of bytes in length, in particular characters, bytes, and booleans, are packed together in memory and then space is left, if necessary, between the end of that declaration and the next:

EXAMPLE:

```
A     : INTEGER;
B     : CHAR;
I,J,K : BYTE;
L     : INTEGER;
```

STORAGE LAYOUT:

```
+0 A LSB (or MSB if Z8000/68000)
+1 A MSB (or LSB if Z8000/68000)
+2 B
```

8080/Z80/6809/8086	Z8000/68000
+3 K	+3 empty space
+4 J	+4 K
+5 I	+5 J
+6 L LSB	+6 I
+7 L MSB	+7 empty space
	+8 L MSB
	+9 L LSB

Structured data types: ARRAYs, RECORDs and SETs require additional explanation:

ARRAYs are stored in ROW major order. This means that A: ARRAY [1..3,1..3] OF CHAR stored as:

```
+0 A[1,1]
+1 A[1,2]
+2 A[1,3]

+3 A[2,1]
+4 A[2,2]
+5 A[2,3]
```

```
+6 A[3,1]
+7 A[3,2]
+8 A[3,3]
```

This is logically a one dimensional array of vectors. In Pascal/MT+ all arrays are logically one dimensional arrays of some other type.

RECORDs are stored in the same manner as global variables.

SETs are always stored as 32 byte items. Each element of the set is stored as one bit. SETs are byte oriented and the low order bit of each byte is the first bit in that byte of the set. Shown below is the set 'A'..'Z':

Byte number

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 ... 1F
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --     --
00 00 00 00 00 00 00 00 FE FF FF 07 00 00 00 00 00 ... 00
```

The first bit is bit 65 ($41) and is found in byte 8 bit 1. The last bit is bit 90 and is found in byte 11 bit 2. In this discussion bit 0 is the least significant bit in the byte.

13.4 Parameter passing

When calling an assembly language routine from Pascal or calling a Pascal routine from assembly language parameters are passed on the stack. Upon entry to the routine the top of the stack contains the return address. Underneath the return address are the parameters in reverse order from declaration: (A,B:INTEGER; C:CHAR) would result in C on top of B on top of A. Each parameter requires at least one 16-bit WORD of stack space. A character or boolean is passed as a 16-bit word with a high order byte of 00. VAR parameters are passed by address. The address represents the byte of the actual variable with the lowest memory address.

Non-scalar parameters (excluding SETs) are always passed by address. If the parameter is a value parameter then code is generated by the compiler in a Pascal routine to call @MVL to move the data. SET parameters are passed by value on the stack and the @SS2 routine is used to store them away.

The example below shows a typical parameter list at entry to a procedure:

```
PROCEDURE DEMO(I,J : INTEGER; VAR Q:STRING; C,D:CHAR);

AT ENTRY STACK:     +0    RETURN ADDRESS
                    +1    RETURN ADDRESS
```

```
+2    D
+3    BYTE OF ØØ
+4    C
+5    BYTE OF ØØ
+6    ADDRESS OF ACTUAL STRING
+7    ADDRESS OF ACTUAL STRING
+8    J (LSB ON 8Ø8Ø, MSB ON Z8k, 68k)
+9    J (MSB ON 8Ø8Ø, LSB ON Z8k, 68k)
+1Ø   I (same as J)
+11   I (same as J)
```

SETs are stored on the stack with the least significant byte on top (low address) and the most significant byte on bottom (high address). Function values are returned on the stack. They are placed "logically" underneath the return address before the return is executed. They therefore remain on the top of the stack after the calling program is re-entered after the return.

Users who wish to call routines written in alien (to the Pascal environment) languages such as PL/I or FORTRAN should observe the following rules:

a) All parameters should be VAR (address)

b) Declare the alien routine in the other language as an EXTERNAL WORD.

c) Declare a local variable which will hold the address of the alien routine.

d) Using the ADDR function take the addr of the EXTERNAL WORD and assign it to the local variable.

e) Call an assembly language routine passing all the parameters and the local variable containing the address of the alien routine.

f) This assembly language routine should remove the addresses from the stack and create a parameter list compatible with the alien language.

g) The assembly language routine should then use the address passed in the WORD to actually call the alien routine.

h) The user should beware of mixing programs which deal with REAL numbers as the format for the reals is likely to be significantly different between the two languages.

i) The user should also beware of assumptions the alien language system makes about who owns what memory resources such as PL/I ALLOCATE and Pascal

HEAP space.

13.5 Restrictions

The user should beware of the following restrictions that are placed upon program which are linkable with the Link/MT+ linker:

a) COMMON is not supported

b) Use of assembly language which would generate external + offset records (e.g. LXI H,EXTVAR+1) should be avoided

c) Use of DB and DW in the DSEG of an assembly language routine is not supported. Use the DB and DW in the CSEG for non-ROM based applications.

d) Use of the Request Library search feature is not supported by Link/MT+.

14.0 Run-time error handling

The Pascal/MT+ system supports two types of run-time checking: range and exception. Range checking is performed on array subscripts and on subrange assignments. The default condition of the system is that these checks are disabled. The user may enable them around any section of coding desired using the $R and $X toggles (see section 2.5 of the applications guide). This section describes the implementation of this mechanism and how users may take advantage of this mechanism to handle run-time errors in a non-standard manner.

The general philosophy is that error checks and error routines will set boolean flags. These boolean flags along with an error code will be loaded onto the stack and the built-in routine @ERR is called with these two parameters. The @ERR routine will then test the boolean parameter. If it is false then no error has occurred and the @ERR routine will exit back to the compiled code and execution continues. If it is true the @ERR routine will print an error message and allow the user to continue or abort.

Listed below are the error numbers passed to the @ERR routine:

Value	Meaning
1	Divide by Ø check
2	Heap overflow check
3	String overflow check
4	Range check

14.1 Range checking

When range checking is enabled the compiler generates calls to @CHK for each array subscript and subrange assignment. The @CHK routine leaves a boolean on the stack and the compiler generates calls to @ERR after the @CHK call. If range checking is disabled and a subscript falls outside the valid range, unpredictable results will occur. For subrange assignments the value will be truncated at the byte level.

14.2 Exception checking

When exception checking is enabled the compiler will load the error flags (zero divide, string overflow, and heap overflow) as needed and call the @ERR routine after each operation which could set the flags. If exception checking is disabled the run-time routines attempt to provide a friendly action if possible: divide by zero results in a maximum value being returned, heap overflow does nothing and string overflow truncates.

14.3 User supplied handlers

It is possible for the user to write an @ERR routine to be used instead of the system routine. The user should declare the routine as:

```
PROCEDURE @ERR(ERROR:BOOLEAN; ERRNUM:INTEGER);
```

The routine will be called, as mentioned above, each time an error check is needed and this routine should check the ERROR variable and exit if it is FALSE. The user may decide the appropriate action if the value is true. The values of ERRNUM are as show in section 14.0

16.0 Appendices

16.1 Error messages

1: Error in simple type
Self-explanatory.

2: Identifier expected
Self-explanatory.

3: 'PROGRAM' expected
Self-explanatory

4: ')' expected
Self-explanatory

5: ': ' expected
Possibly a = used in a VAR declaration

6: Illegal symbol (possibly missing ';' on line above)
Symbol encountered is not allowed in the syntax at this point.

7: Error in parameter list
Syntactic error in parameter list declaration.

8: 'OF' expected
Self-explanatory.

9: '(' expected
Self-explanatory.

10: Error in type
Syntactic error in TYPE declaration.

11: '[' expected
Self-explanatory.

12: ']' expected
Self-explanatory.

13: 'END' expected
All procedures, functions, and blocks of statements must have an 'END'. Check for mismatched BEGIN/ENDs.

14: ';' expected (possibly on line above)
Statement separator required here.

15: Integer expected
Self-explanatory.

16: '=' expected
Possibly a : used in a TYPE or CONST declaration.

17: 'BEGIN' expected
Self-explanatory.

18: Error in declaration part
Typically an illegal backward reference to a type in a pointer declaration.

19: error in <field-list>
Syntactic error in a record declaration

20: '.' expected
Self-explanatory.

21: '*' expected
Self-explanatory.

50: Error in constant
Syntactic error in a literal constant

51: ':=' expected
Self-explanatory.

52: 'THEN' expected
Self-explanatory.

53: 'UNTIL' expected
Can result from mismatched begin/end sequences

54: 'DO' expected
Syntactic error.

55: 'TO' or 'DOWNTO' expected in FOR statement
Self-explanatory.

56: 'IF' expected
Self-explanatory.

57: 'FILE' expected
Probably an error in a TYPE declaration.

58: Error in <factor> (bad expression)
Syntactic error in expression at factor level.

59: Error in variable
Syntactic error in expression at variable level.

99: MODEND expected
Each MODULE must end with MODEND.

101: Identifier declared twice
Name already in visible symbol table.

102: Low bound exceeds high bound
For subranges the lower bound must be <= high bound.

103: Identifier is not of the appropriate class
A variable name used as a type, or a type used as a variable, etc. can cause this error.

104: Undeclared identifier
The specified identifier is not in the visible symbol table.

105: sign not allowed
Signs are not allowed on non-integer/non-real constants.

106: Number expected
This error can often come from making the compiler totally confused in an expression as it checks for numbers after all other possibilities have been exhausted.

107: Incompatible subrange types
(e.g. 'A'..'Z' is not compatible with 0..9).

108: File not allowed here
File comparison and assignment is not allowed.

109: Type must not be real
Self-explanatory.

110: <tagfield> type must be scalar or subrange
Self-explanatory.

111: Incompatible with <tagfield> part
Selector in a CASE-variant record is not compatible with the <tagfield> type

112: Index type must not be real
An array may not be declared with real dimensions

113: Index type must be a scalar or a subrange
Self-explanatory.

114: Base type must not be real
Base type of a set may be scalar or subrange.

115: Base type must be a scalar or a subrange
Self-explanatory.

116: Error in type of standard procedure parameter
Self-explanatory.

117: Unsatisified forward reference
A forwardly declared pointer was never defined.

118: Forward reference type identifier in variable declaration
The user has attempted to declare a variable as a pointer to a type which has not yet been declared.

119: Re-specified params not OK for a forward declared procedure
Self-explanatory.

120: Function result type must be scalar, subrange or pointer
A function has been declared with a string or other non-scalar type as its value. This is not allowed.

121: File value parameter not allowed
Files must be passed as VAR parameters.

122: A forward declared function's result type can't be re-specified
Self-explanatory.

123: Missing result type in function declaration
Self-explanatory.

125: Error in type of standard procedure parameter
This is often caused by not having the parameters in the proper order for built-in procedures or by attempting to read/write pointers, enumerated types, etc.

126: Number of parameters does not agree with declaration
Self-explanatory.

127: Illegal parameter substitution
Type of parameter does not exactly match the corresponding formal parameter.

128: Result type does not agree with declaration
When assigning to a function result, the types must be compatible.

129: Type conflict of operands
Self-explanatory.

130: Expression is not of set type
Self-explanatory.

131: Tests on equality allowed only
Occurs when comparing sets for other than equality.

133: File comparison not allowed
File control blocks may not be compared as they contain multiple fields which are not available to the user.

134: Illegal type of operand(s)
The operands do not match those required for this operator.

135: Type of operand must be boolean
The operands to AND, OR and NOT must be BOOLEAN.

136: Set element type must be scalar or subrange

Self-explanatory.

137: Set element types must be compatible
Self-explanatory.

138: Type of variable is not array
A subscript has been specified on a non-array variable.

139: Index type is not compatible with the declaration
Occurs when indexing into an array with the wrong type of indexing expression.

140: Type of variable is not record
Attempting to access a non-record data structure with the 'dot' form or the 'with' statement.

141: Type of variable must be file or pointer
Occurs when an up arrow follows a variable which is not of type pointer or file.

142: Illegal parameter solution
Self-explanatory.

143: Illegal type of loop control variable
Loop control variables may be only local non-real scalars.

144: Illegal type of expression
The expression used as a selecting expression in a case statement must be a non-real scalar.

145: Type conflict
Case selector is not the same type as the selecting expression.

146: Assignment of files not allowed
Self-explanatory.

147: Label type incompatible with selecting expression
Case selector is not the same type as the selecting expression.

148: Subrange bounds must be scalar
Self-explanatory.

149: Index type must be integer
Self-explanatory.

150: Assignment to standard function is not allowed
Self-explanatory.

151: Assignment to formal function is not allowed
Self-explanatory.

152: No such field in this record
Self-explanatory.

153: Type error in read

Self-explanatory.

154: Actual parameter must be a variable
Occurs when attempting to pass an expression as a VAR paraeter.

155: Control variable cannot be formal or non-local
The control variable in a FOR loop must be LOCAL.

156: Multidefined case label
Self-explanatory.

157: Too many cases in case statement
Occurs when jump table generated for case overflows its bounds.

158: No such variant in this record
Self-explanatory.

159: Real or string tagfields not allowed
Self-explanatory.

160: Previous declaration was not forward

161: Again forward declared

162: Parameter size must be constant

163: Missing variant in declaration
Occurs when using NEW/DISPOSE and a variant does not exist.

164: Substition of standard proc/func not allowed

165: Multidefined label
Label more than one statement with same label.

166: Multideclared label
Declare same label more than once.

167: Undeclared label
Label on statement has not been delcareed.

168: Undefined label
A declared label was not used to label a statement.

169: Error in base set

170: Value parameter expected

171: Standard file was re-declared

172: Undeclared external file

174: Pascal function or procedure expected
Self-explanatory.

183: External declaration not allowed at this nesting level
Self-explanatory.

187: Attempt to open library unsuccessful
Self-explanatory.

191: No private files
Files may not be declared other than in the GLOBAL variable section of a program or module as they must be statically allocated.

193: Not enough room for this operation
Self-explanatory.

194: Comment must appear at top of program

201: Error in real number - digit expected
Self-explanatory.

202: String constant must not exceed source line

203: Integer constant exceeds range
Range on integer constants are -32768..32767

250: Too many scopes of nested identifiers
There is a limit of 15 nesting levels at compile-time. This includes WITH and procedure nesting.

251: Too many nested procedures or functions
There is a limit of 15 nesting levels at execution time.

253: Procedure too long
A procedure has generated code which has overflowed the internal procedure buffer. Reduce the size of the procedure and try again. The limit is target machine dependent. Consult the CPU applications note for more information.

259: Expression too complicated
The users expression is too compilated (i.e. too many recursive calls needed to compile it). The user should reduce the compilcation using temporary variable

397: Too many FOR or WITH statments in a procedure
Only 16 FOR and / or WITH statments are allowed in a single procedure (in recursive mode only)

400: Illegal character in text
A character which is a non-Pascal special character was found outside of a quoted string.

401: Unexpected end of input
End. encountered before returning to outer level.

402: Error in writing code file, not enough room
Self-explanatory.

403: Error in reading include file
Self-explanatory.

404: Error in writing list file, not enough room
Self-explanatory.

405: Call not allowed in separate procedure
Self-explanatory.

406: Include file not legal
Self-explanatory.

407: Symbol Table Overflow

497: Error in closing code file.
An error occured when the .ERL file was closed.
Make more room on the destination disk and try again.

16.2 Reserved Words

The following are the reserved words in Pascal/MT+:

MOD, NIL, IN, OR, AND, NOT, IF, THEN, ELSE,
CASE, OF, REPEAT, UNTIL, WHILE, DO, FOR, TO,
DOWNTO, BEGIN, END, WITH, GOTO, CONST, VAR,
TYPE, ARRAY, RECORD, SET, FILE, FUNCTION,
PROCEDURE, LABEL, PACKED, PROGRAM

Pascal/MT+ also has extended reserved words:

ABSOLUTE, EXTERNAL

16.3 Language syntax description

```
<letter> ::= A | B | C | D | E | F | G | H | I | J |
             K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z | a | b | c | d |
             e | f | g | h | i | j | k | l | m | n |
             n | o | p | q | r | s | t | u | v | w |
             x | y | z | @

<digit>  ::= Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
             A | B | C | D | E | F   {only allowed in HEX numbers}

<special symbol > ::= {reserved words are listed in section 16.2}
             +  |  -  |  *  |  /  |  =  |  <> |  <  |  >  |  .
             <= |  >= |  (  |  )  |  [  |  ]  |  {  |  }  |
             := |  .  |  ,  |  ;  |  :  |  '  |  ^  |

             {the following are additional or substitutions:}
             (. |  .) |  ~  |  \  |  ?  |  !  |  |  |  $  |  &

             (. is a synonym for [
             .) is a synonym for ]
             ~, \, and ? are synonyms  (see section 8.1.1)
             !, and | are synonyms     (see section 8.1.2)
             &                         (see section 8.1.3)

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

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

<digit sequence>      ::= <digit> {<digit>}

<unsigned integer>    ::= $ <digit sequence> |
                          <digit sequence>

<unsigned real>       ::= <unsigned integer> . <digit sequence>    |
                          <unsigned integer> . <digit sequence>
                          E <scale factor>                         |
                          <unsigned integer> E <scale factor>

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

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

<sign>                ::= + | -

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

<constant identifier> ::= <identifier>

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

```
                          <constant identifier>       |
                          <sign><constant identifier> |
                          <string>

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


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

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

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

<type identifier> ::= <identifier>

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

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

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

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

<array type>     ::= <normal array> |
                     <string array>>

<string array>   ::= STRING <max length>

<max length>     ::= [ <intconst> ] |
                     <empty>

<intconst>       ::= <unsigned integer> |
                     <int const id>

<int const id>   ::= <identifier>

<normal array>   ::= ARRAY [ <index type> {,<index type>}] OF
                     <component type>

<index type>     ::= <simple type>

<component type> ::= <type>

<record type>     ::= RECORD <field list> END

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

```
                        <variant part>

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

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

<variant part>      ::= CASE <tag field> <type identifier> OF
                        <variant> {;<variant>}

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

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

<case label>        ::= <constant>

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

<set type>   ::= SET OF <base type>

<base type>  ::= <simple type>

<file type>  ::= file {of <type>}

<variable>      ::= <var>                |
                    <external var>       |
                    <absolute var>

<external var>  ::= EXTERNAL <var>

<absolute var>  ::= ABSOLUTE [ <constant> ] <var>

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

        Declaration of variables of type STRING:

        <identifier>{,<identifier>} : STRING {[<constant>]}

<entire variable>      ::= <variable identifier>

<variable identifier>  ::= <identifier>

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

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

<array variable>    ::= <variable>

<field designator>  ::= <record variable> . <field identifier>
```

```
<record variable>  ::= <variable>

<field identifier> ::= <identifier>

<file buffer>   ::= <file variable> ^

<file variable> ::= <variable>

<referenced variable> ::= <pointer variable> ^

<pointer variable>    ::= <variable>

<unsigned constant> ::= <unsigned number>
                        <string>
                        NIL
                        <constant identifier>

<factor>            ::= <variable>
                        <unsigned constant>
                        <function designator>
                        ( <expression> )
                        <logical not operator> <factor>

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

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

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

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

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

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

<logical not operator> ::= NOT  |  ~  |  \  |  ?

        ~ (synonyms \ and ?) is a NOT operator for non-booleans.

<multiplying operator> ::= *  |  /  | DIV | MOD | AND | &

        & is an AND operator on non-booleans.

<adding operator> ::= +  |  -  |  OR  |  |  |  !

        ! (synonym |) is an OR operator on non-booleans.

<relational operators> ::=  =  |  <>  |  <  |  <=  |  >  |  >=  |  IN
```

```
<function designator> ::= <function identifier>                          |
                          <function identifier> ( <parm> {,<parm> )

<function identifier> ::= <identifier>

<statement>           ::= <label> : <unlabelled statement> |
                          <unlabelled statement>

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

<label>               ::= <unsigned integer>

<simple statement ::= <assignment statement> |
                      <procedure statement>  |
                      <goto statement>       |
                      <empty statement>

<empty statement> ::= <empty>

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

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

                          <procedure identifier>

<procedure identifier>::= <identifier>

<parm>                ::= <procedure identifier> |
                          <function identifier>  |
                          <expression>           |
                          <variable>

<goto statement> ::= goto <label>

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

<compound statement> ::= BEGIN <statement> {,<statement>} END

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

<if statement> ::= IF <expression> THEN <statement> ELSE <statement> |
                   IF <expression> THEN <statement>

<case statement> ::= CASE <expression> OF
                     <case list> {,<case list>}
                     {ELSE <statement>}
                     END
```

```
<case list>       ::= <label list> : <statement> |
                      <empty>

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

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

<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> {,<statement>} UNTIL <expression

<for statement> ::= FOR <ctrlvar> := <for list> DO <statement>

<for list>        ::= <expression> DOWNTO <expression> |
                      <expression>   TO   <expression>

<ctrlvar>         ::= <variable>

<with statement>        ::= WITH <record variable list> DO <statment>

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

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

<block>                      ::= <label declaration part>
                                 <constant definition part>
                                 <type definition part>
                                 <variable declaration part>
                                 <procfunc declaration part>
                                 <statement part>

<procedure heading>          ::= PROCEDURE <identifier> <parmlist>    |
                                 PROCEDURE <identifier> ;             |
                                 PROCEDURE INTERRUPT [ <constant> ] ;

<parmlist>                   ::= ( <fparm> {,<fparm>} )

<fparm>                      ::= <procedure heading> |
                                 <function heading>  |
                                 VAR <parm group>    |
                                 <parm group>

<parm group>                 ::= <identifier> {,<identifier>} :
                                   <type identifier>           |
                                 <identifier> {,<identifier>} :
                                   <conformant array>

<conformant array>           ::= ARRAY [ <indxtyp> {;<indxtyp} ] OF
                                 <conarray2>

<conarray2>                  ::= <type identifier>   |
                                 <conformant array>
```

```
<indxtyp>                    ::= <identifier> .. <identifier> : <ordtypid>

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

<label declaration part>     ::= <empty> |
                                 LABEL <label> {,<label>} ;

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

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

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

<procfunc part>              ::= {<proc or func> ; }

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

<statement part>             ::= <compound statement>

<function decl>    ::= EXTERNAL <function heading> |
                       <function heading> <block>

<functon heading>  ::= FUNCTION <identifier> <parmlist> : <result type> ; |
                       FUNCTION <identifier> : <result type> ;

<result type>      ::= <type identifier>

<readcall>         ::= <read or readln> {( {<filevar> ,} {<varlist>} )}

<read or readln>   ::= READ | READLN

<filevar>          ::= <variable>

<varlist>          ::= <variable> {,<variable>}

<writecall>          ::= <write or writeln> {( {<filevar> ,} {exprlist} )}

<write or writeln>   ::= WRITE | WRITELN

<exprlist>           ::= <wexpr> {,<wexpr}

<wexpr>              ::= <expression> {:<width expr> {:<dec expr>}}
```

```
<width expr>        ::= <expression>

<dec expr>          ::= <expression>

<program>           :=  <program heading> <block> . |
                        <module  heading>
                           <label declaration part>
                           <constant definition part>
                           <type definition part>
                           <variable declaration part>
                           <procfunc declaration part>
                           MODEND .

<program heading>  ::= PROGRAM <identifier> ( <prog parms> ) ;

<module heading>   ::= MODULE <identifier> ;

<prog parms>       ::= <identifier> {,<identifier>}
```

16.4 Summary of option switches and toggles

Compiler command line: (see section 2.0)

```
MTPLUS <filename> {$ option switches}
```

Compiler switches: (see section 2.0)

Rd	route .ERL file to disk d:
nd	get .OVL file #n from disk d: (n=1..4)
Pd	route .PRN file to disk d:
X	generate disassembler records in .ERL file
D	generate debugger calls and .PSY file
Ed	get MTERRS.TXT file from disk d:
Td	put PASTEMP.TOK (temporary) file on disk d:
Q	Quiet option
A	Auto chain to linker
B	Use BCD reals
Z	Generate Z80 code (8080/Z80 version only)

Compiler toggles: (see section 2.5)

$E	controls entry point generation
$S	controls recursion
$I	include file control
$R	controls range checking
$T	controls strict type checking
$W	controls non-ISO warnings
$X	controls exception checking
$P	inserts formfeed in .PRN file
$L	controls listing (on/off)
$K	allows removal of pre-defined routines from symbol table to save memory space
$C	controls use of RST instructions in REAL operations

Linker command line: (see section 3.0)

```
LINKMT <filename> {,<filename} {option switches}

or
```

```
LINKMT <filename>=<filename> {,<filename>} {option switches}
```

Linker switches: (see section 3.1)

/S	library search
/L	load display
/M	load map table display
/E	extended /M (includes ?, $, @)
/P:nnnn	org program at nnnnH
/D:nnnn	org data at nnnnH
/H:nnnn	write .HEX file addresses start with nnnnH
/W	write .SYM file
/F	previous file is a command file

Disassembler command line: (see section 10.0)

```
DIS???? <input name> {<destination name> {,L=nnn}}
```

Debugger commands: (see section 11.0)

DV	Display variable
DI	Display integer
DC	Display character
DL	Display logical (boolean)
DR	Display real
DB	Display byte
DW	Display word
DS	Display string
DX	Display extended
TR	trace one line
Tn	trace n lines
BE	begin exec at main program
GO	continue after breakpoint
SB	set breakpoint
RB	remove breakpoint
E+	entry/exit display on
E-	entry/exit display off
PN	display all procedure names
VN	display variable names
??	HELP! - display command summary

Pascal/MT+ Release 5.1

User's Guide Addenda

November 10, 1980

Contents:

Section I Introduction

This manual addendum contains a number of very important notes regarding Pascal/MT+ 5.1. The user should READ CAREFULLY all sections of this document BEFORE USING Pascal/MT+.

We have strived to make Pascal/MT+ one of the finest software tools available and we are convinced that we have attained this goal. We welcome any comments and constructive criticism regarding Pascal/MT+. If you have any problems please let us know as soon as possible.

This document contains notes on the status of the Pascal/MT+ product with regards to design goals (i.e. memory space). In addition, documentation on new features added since the manual was printed and documentation on a few minor features which were implemented in a different way than specified in the printed manual is also included.

Good luck with Pascal/MT+ and happy Pascal programming!

Section II Released software status

The enclosed software (if you have purchased more than just the manual) will run in a MINIMUM 52K CP/M SYSTEM. This means: IN A SYSTEM WITH A STANDARD SIZE BIOS OF <= 768 BYTES AND A STANDARD CP/M 1.3/1.4/2 BDOS (4K-5K bytes) AND 52K OF CONTIGUOUS MEMORY, THE SYSTEM SHOULD OPERATE.

Your BDOS (not BIOS) should be no lower in memory than address OBC00H. You can check this by loading in DDT (or DEBUG if you have CDOS) and listing location 0005 (command:L5). The address shown for the jump instruction will be the address of a jump to some code which will eventually jump to the BDOS. List the address shown as the operand of the jump instruction and you should find you way easily from there. This means a TPA (transient program area) size of (52K-5K=47K) is required!!!!! We recommend that any attempts at large program development be done in a system with at least 56K and the more the merrier. The system will dynamically adapt to more memory if available.

The compiler/library has been put through the Pascal validation suite and the results are attached in the validation suite report. We have been using the compiler to compile itself, assemblers, editors, linkers, etc. for the last couple of months and are confidant of stable operation. Please, if you suspect a problem USE THE DISASSEMBLER to verify that the compiler is generating incorrect code before calling! In addition: we KINDLY REQUEST! that you have the following information handy before calling!!!:

CP/M VERSION (1.3/1.4/2.0/2.1/2.2 OR equivalent information)

MEMORY SIZE

CPU TYPE (8080 / Z80 / 8085, ETC.)

YOUR PASCAL/MT+ SERIAL NUMBER AND APPROXIMATE DATE PURCHASED

YOUR COMPANY NAME

DIASSEMBLED LISTING OF THE PROGRAM SECTION IN QUESTION

AND REMEMBER: IF YOU HAVE NOT SENT IN YOUR LICENSE AGREEMENT DON'T CALL, WE WILL NOT HELP YOU!!!!!

Section III Functional Additions

Since the Pascal/MT+ manual was printed we have modified a number of minor implementation details and have also added a number of features. The list below describes each in detail:

WORD VARIABLE INPUT/OUTPUT

1. WORD I/O as described on page 35 of the manual is not implemented using READ and WRITE. Two new procedures:

```
READHEX (VAR F:TEXT; VAR W:ANYTYPE; SIZE:1..2);

WRITEHEX(VAR F:TEXT; EXP : ANYTYPE; SIZE:1..2);
```

   have been implemented allowing HEX I/O on variables of any 1 or two byte type such as integer, char, byte, subrange, enumerated and word.

RE-DIRECTED I/O OF STRINGS

2. The use of READ and STRING variables is not allowed when re-directed I/O is used. This is because the @RST routine attempts to read directly from the CP/M console device when no file is specified. The user should re-write the @RST routine to perform any and all input and editing functions desired for the target system console device. NOTE: THIS DOES NOT AFFECT PROGRAMS DO NOT USE RE-DIRECTED I/O

USE OF # ON COMPILER COMMAND LINE

3. The compiler also allows the use of the # character as the option string signal character on the command line. This is because the CP/M SUBMIT program does not allow the user to place strings with $ in them in the submit file. With other software the user must place a dummy parameter and then put this parameter on the command line when calling submit even if this is not really a variable parameter to the submit file. Using the alternate form

(e.g. #PC X RB) allows insertion of this directly into the .SUB file

"LOCAL" "TEMPORARY" FILES IN A RECURSIVE ENVIRONMENT

4. Locally declared files in a recursive environment may not be used as "temporary" files unless the user explicitly zeroes the file (using FILLCHAR) or does an ASSIGN(<file>,'') to initalize the file. The user should also note that such locally declared files will be left open and in limbo when the procedure is exitted unless explictly CLOSEd.

PHASE 2 CONSOLE OUTPUT

5. The output produced by Phase 2 of the compiler consists of the Procedure or Function name and its offset from the beginng of the module in decimal. This is output when the procedure/function body is actually encountered (i.e. if A contains B which contains C then the output would be C followed by B followed by A).

USE OF TRM: DEVICE

6. Non-echo input (TRM:) is only operative on CP/M version 2.x systems

USE OF WRITE/WRITELN WITH FUNCTIONS WHICH PERFORM I/O

7. The user should not use the WRITE/WRITELN procedures to output the value of any functions which operate on files (such as GNB) because the file pointers will become modified by the reading routines and therefore the output will suddenly be done to the input file!

NON-TEXT FILE END-OF-FILE HANDLING

8. Because CP/M does not keep any information regarding partially filled sectors at the end of a non-TEXT file it is impossible to make EOF(<non-text-file>) work properly unless the size of the record is exactly a multiple of 128 bytes. The suggested way of working around this operating system problem is to keep a count of the number of records in the file or have a special end-of-file record.

NEW COMPILER OVERLAYS (MT185.OVL AND MT580.OVL)

9. Two new overlay files are present. MT185.OVL was created by breaking the previously large Phase 1 and initialization into two separate overlays (MT180 and MT185). MT580.OVL has been added to write the .PSY file for the debugger. The location of this file is controlled by the OVL #2 option switch. Note: MT580.OVL is not necessary (and never loaded) unless the debugger is requested.

HEAP MANAGEMENT ROUTINES - FULL VERSUS STACK

10. The run-time library contains a "stack" oriented HEAP management module which supplies only NEW and DISPOSE. A separate module (FULLHEAP.ERL) is supplied which fully implements NEW / DISPOSE / MEMAVAIL / MAXAVAIL and must be explicitly named on the command line when desired. PASLIB was getting too big to keep on adding and adding and...

NEW ERROR MESSAGES

11. Three new errors have been added:

    496 - invalid operand to INLINE

    398 - Implementation restriction (normally used for arrays and sets which are too big to be manipulated or allocated)

999 - Compiler Totally Confused

REGARDING AUTOMATIC CLOSING OF OUTPUT FILES

12. On page 95 of the user's guide the manual discusses automatic closing of OPEN files. This has been eliminated from the package for the time being due to a desire not to force ROM based users to include the ENTIRE FILE PACKAGE in their programs without re-writing a large portion of the run-time. This feature may be added in a later release.

UTILITY MODULE

13. A file called UTILMOD.SRC is on the distribution disk and contains three routines:

```
FUNCTION  KEYPRESSED : BOOLEAN;
(* returns true when a key struck *)

PROCEDURE EXTRACT(VAR F:TEXT; VAR S:STRING);
(* extracts the file name string from an open file *)

PROCEDURE RENAME(VAR F:TEXT; NEWNAME:STRING);
(* used after an assign to change the name of a file *)
```

PATCHER PROGRAM

14. The PATCHER.COM program has been eliminated.

NEW $K TOGGLES IMPLEMENTED

15. Additional $K toggles have been implemented.
(note a separate $K switch is required for each group)

Group	Routines Removed
8	RESET, REWRITE, GET, PUT, ASSIGN MOVELEFT, MOVERIGHT, FILLCHAR

9	READ, READLN
10	WRITE, WRITELN
11	TSTBIT, CLRBIT, SETBIT, SHL, SHR
12	MEMAVAIL, MAXAVAIL
13	SEEKREAD, SEEKWRITE
14	RIM85, SIM85, WAIT
15	READHEX, WRITEHEX

8085 RIM AND SIM

16. The pre-defined I/O names RIM85 and SIM85 have been changed to a function and a procedure:

```
FUNCTION RIM85 : BYTE;

PROCEDURE SIM85 (VAL: BYTE);
```

17.
```
**********************************************
*                                            *
* R A N D O M   A C C E S S   F I L E   I / O *
*                                            *
**********************************************
```

Since the manual was printed we settled on how random access files will work:

a. You must have CP/M version 2
b. A random access file and a sequential file are not compatible because:

   a sequential is totally a stream of bytes
   a random file is allocated as follows:

   if the recordsize is < 128 bytes then as many records as will fit in a sector will be packed into a sector and the remainder of the sector left garbage

   if the recordsize is >= 128 bytes then the record will begin on a sector boundary and be allocated an integral number of sectors (i.e. records do not begin in the middle of a sector but will span sectors as necessary)

c. Random access files are accessed via:

```
PROCEDURE SEEKREAD (VAR F:ANYFILETYPE; RECNUM:INTEGER);

PROCEDURE SEEKWRITE(VAR F:ANYFILETYPE; RECNUM:INTEGER);
```

Recnum starts at 0
IORESULT contains the CP/M return code for the access
Both these procedures work with the file window variable (e.g. F^). The user should assign to the window variable before SEEKWRITE and assign from the window variable after a SEEKREAD.

THE USER IS REFERRED TO THE TESTRIO.SRC PROGRAM ON THE DISTRIBUTION DISK!

MEMORY LAYOUT UNDER CP/M

18. Users have requested to know how to find the end of the Pascal allocated data area. In Pascal/MT+ there is a system variable SYSMEM which at run-time points to the end of the allocated data. Even in programs which use the FULLHEAP module the SYSMEM pointer is updated. The run-time evaluation stack is pointed to by the hardware SP register and the recursion stack is pointed to by a variable called @SFP (stack frame pointer). Both SYSMEM and @SFP may be declared as EXTERNAL WORD and accessed by the user for special requirements.

RUN-TIME PACKAGE SOURCE CODE

19. The run-time package source code comes in four groups of files (.LIB and .IDX):

CPMIO - Pascal routines for file input/output

RTP - Assembly language support routines (use MACRO-80 or RMAC to assemble)

FP - Floating point run time routines

BCD - BCD real routines

To extract a given module from these "libraries" the user should find the desired name in the

index and then use the SPLIT program to extract the modules. The SPLIT program expects a "library" file and a file containing a list of file names which are to be extracted. The library will be and individual files will be created for the requested files.

AMD9511 HARDWARE ARITHMETIC!!

To use the AMD9511 the user must use DECCONV.LIB, XCONFIG.LIB and FP.MAC Edit the XCONFIG.LIB file and change the HARDWARE equate to TRUE and set ADATA and ACTRL to the I/O port addresses for the 9511.

Then assemble FP.MAC and this will create a new FP.REL which shoule be combined with the supplied PFLT.ERL and then renamed FPREALS.ERL. The user can combine them using MTLIB.COM and the COMBINE.CMD file. Now hardware floating point may be linked with programs.

When using the 9511 the user may wish to declare and call @I95 which is a parameterless procedure which initializes the 9511 chip. Some old 9511 chips did not properly reset using the hardware reset line and this software routine will convince the 9511 that all is ok and ready to go. This should be called as the first statement of the main program.

ISO STANDARD

20. Users who wish to receive a copy of the proposed ISO standard to which we have been working should contact MT MicroSYSTEMS. We have a limited number of copies available for $20.00. Only users who are very fluent in Pascal and compilers should be interested as the standard document is very terse and sometimes very confusing!

BCD AND FLOATING POINT REAL CONSIDERATIONS

21. USERS SHOULD NOTE: WHEN USING REAL NUMBERS EITHER

BCDREALS OR FPREALS MUST BE LINKED WITH THE PROGRAM BEFORE PASLIB. ALSO IF TRANSCENDENTALS ARE USED IN FLOATING POINT PROGRAMS "TRANCEND" SHOULD BE LINKED BEFORE FPREALS. ALSO NO SQRT ROUTINE IS AVAILABLE IN THE BCD REALS PACKAGE.

CP/M COMMAND LINE EXTRACTION

22. To use CP/M command line info the user should declare an absolute variable (PACKED ARRAY [0..127] OF CHAR); with an address of $80 and then move this to a string (STRING[127]). The user should note that the first character of this string will typicaly contain a blank.

```
VAR
  CPMCMDBUF : ABSOLUTE [$80] PACKED ARRAY [0..127] OF CHAR;
  CPMCMDSTR : STRING[127];

BEGIN
  MOVE(CPMCMDBUF,CPMCMDSTR,128);
  ...
END.
```

USE OF RESTARTS FOR REALS AND RECURSION

23. In a similar manner to the $C toggle used for calling the @XOP routine for real numbers using restarts, the $Q toggle has been added to perform the same operation with recursive modules. Every call to the @DYN routine will be converted to a restart n (where n is the parameter to $Q (e.g. ($Q 5)). NOTE: FOR BOTH THIS FEATURE AND $C THE SWITCH MUST BE BOTH IN MODULES AND IN THE MAIN PROGRAM SO THAT THE RESTART VECTOR LOAD CODE IS GENERATED AT THE BEGINING OF THE MAIN PROGRAM.

Section IV Manual Modifications

Listed below are manual corrections:

1. Page 43 and Page 6 - section 5.22 is on page 97 not 87

2. Page 52 - system now requires 52K (not 48K)

3. Page 72 - second assignment statment should be to S2
   comparison should be S1 < S2
   output should read "is less than" rather than <

4. Page 80 - Add HL=260 after Output:

5. Page 21 - Remove the reserved word NOT from syntax
   and add <logical not operator>

6. Page 32 - add GNB/WNB to list of file proc/func

Section V Validation suite results

The Pascal validation suite (a collection of more than 200 programs) has been run with Pascal/MT+. We have endeavored to pass all of the "conformance" test and have succeeded in all but three cases. Only one of these cases is inherent in the compiler itself, the other two have to do with the precision of our floating point package and output formatting in our floating point. In addition we have discovered 6 programs in the "conformance" section which we believe to actually be incorrect. We have listed below the program names (as found in the suite) and the results of the tests which failed:

Name	Reason failed
6.2.2-3	The standard states that forwardly declared pointers must not reference backwards if there are is a type in the current block with the same name. This is quite compilcated but will probably be fixed in a future version of the compiler
6.6.6.2-3	Our square root routine is not (as is typical with floating point) empirically accurate (e.g. SQRT(25) =4.99999.
6.9.4-4	Our floating point output conversion routine does not match specifically the formats specified by the standard. We will revise this later

**** Errors in the validation suite ****

6.1.2-3	The standard specifies that identifiers may be of any length but does not specify to what degree these identifiers must be unique. This test uses names with uniqueness past the 8th character.
6.4.3.5-1	This test attempted to declare a file of a variable name
6.6.3.1-1	This test does not meet the requirement for identical types in VAR parameter passing
6.6.3.4-2	This test is syntactically incorrect
6.9.4-6	This test wrote 'AAAAA':1 and expected to have only one A output. The standard says that all characters are output.
6.9.4-7	This test writes (TRUE:5,FALSE:5) and expects to get 'TRUE FALSE' not ' TRUEFALSE' as specified by the standard.

6.8.3.5-4 This test attempts to have a CASE statement with selectors of -1000 and +1000. This test should have been included in the implementation defined or quality sections because the standard does not state the required range of case statement selectors which must be accepted