JRT PASCAL 3.0

USERS GUIDE

JRT Systems, 45 Camino Alto, Mill Valley, CA 94941

COPYRIGHT

DISCLAIMER

TRADEMARKS

Supplement to JRT Pascal 3.0 User's Guide

A. Programming Notes

1. All programs and external procedures prepared with JRT Pascal versions 2.x should be recompiled with the new compiler.

2. The external functions ARCTAN, COS, EXP, LN, SIN, SQRT are supplied in source code form and must be compiled before use.

3. Version 3.0 now supports full file variables. File variables may be used as reference parameters (indicated by VAR) but should not be used as value parameters.

4. Dynamic arrays may not be referenced as structures. Only elements of dynamic arrays may be referenced in programs. Fillchar should not be used to initialize dynamic arrays.

5. Dynamic arrays should always be DEALLOCATED before being reallocated to a different size.

6. CP/M 2.2 is required to use the random file facilities. Since Customiz and the Linker use random files, they also require CP/M 2.2.

B. Typographic errors

p. 3-4 List of files should include ERASE.INT, RENAME.INT, VERIFY.INT, READTHIS. The filetype of the external functions should be PAS rather than INT.

p. 107 In the example program TESTPICT, the external reference is:

```
FUNCTION PICTURE ( FMT : STRING; R : REAL ): STRING; EXTERN;
```

p. 149 The number of blocks in the SAVE commands should be 93 for EXEC.COM and 85 for JRTPAS3.COM.

p. 173 The list of Activan commands is incorrect here and on the Reference card. It should be:

```
C    clear the counters
H    display histogram
I    initialize line range
M    run program with monitoring
R    run program without monitoring
Z    terminate the program
```

p. 178 In the declaration of record type jgraf_interface, change the field name "title" to "graf_title".

1. EXTERNAL PROCEDURE DECLARATIONS

One of the most common programming errors reported to us is declaring external procedures in the wrong order within external procedure modules. In the Pascal source program of an external procedure which calls other external procedures, the procedure header of the called procedures always comes after the header of current procedure. That is - only global const, type and var declarations can come between the word extern and the procedure header.

```
EXTERN
{ optional global CONST, TYPE, VAR declarations }
PROCEDURE THISPROC ( X,Y : INTEGER );
VAR
N, M : INTEGER;    { variables local to THISPROC }

PROCEDURE EXPROC1; EXTERN;
PROCEDURE EXPROC2 ( A : REAL ); EXTERN;
FUNCTION EXFUN1 ( X : REAL ): REAL; EXTERN;

BEGIN
{ THISPROC Pascal code }
END; .
```

The external procedure example on the reference card is in error.

2. FILES OF RECORD LENGTH GREATER THAN 128

The procedures GET, PUT and window variables should not be used with files whose record length (declared in the FILE OF ...) is greater than 128 bytes.

Random files with records of any length are allowed. READ/WRITE may be used to sequentially input/output records greater than 128 bytes if one of the following is done:

A. if the file's record size exceeds 128 then it should be declared as FILE OF CHAR
B. install this patch using CP/M's DDT utility program:

```
A>DDT EXEC.COM
DDT VERS 1.4
NEXT  PC   END
5E00 0100 C7FF
-A366E
366E  LXI H,7F
3671  .
-A466E
466E  LXI H,7F
4671  .
-G0
A>SAVE 93 EXEC.COM
```

Table of Contents

JRT Pascal version 3.Ø

This is a major enhancement over earlier versions of JRT Pascal:

version	release date
1.3	March 198Ø
1.4	August 198Ø
2.Ø	January 1982
2.1	July 1982
2.2	November 1982
3.Ø	January 1983

Version 3.Ø includes internal improvements and these major new features:

1. expanded user manual with 3-ring binder
2. JRT Pascal reference card
3. full support for indexed files (7.)
4. CRTMAP utility for full-screen record display (15.)
5. PICTURE external function for number formatting
6. full support for Pascal file variables and GET/PUT (7.)
7. dynamic arrays - ALLOCATE, DEALLOCATE (4.9)
8. SEARCH external function (5.2Ø)
9. %INCLUDE directive (3.4)
10. improved compiler listing, %TITLE, %PAGE(n)

To make use of the new features, programs written for earlier versions should be recompiled under version 3.Ø.

1. Introduction

Pascal is a high level programming language named after the French philosopher and mathematician Blaise Pascal (1623-1662). Nicklaus Wirth developed the language beginning in 1968. It is a descendent of the Algol family of languages which incorporates principles of structured programming.

JRT Pascal was designed specifically for the CP/M operating system. It includes many state of the art features not before available in any microcomputer language.

1.1 JRT Pascal features

With JRT Pascal, programs of practically unlimited size can be developed. External procedures and functions written in Pascal or assembly language are separately compiled. They are automatically loaded from disk when they are first referenced or they may be merged with the main program to form one module. The advanced dynamic storage system will purge infrequently used procedures if storage becomes full. Dynamic storage compression ensures the optimum use of the main storage resource.

The floating point arithmetic provides 14 digits of precision. All standard functions are supported.

The input/output system supports sequential and two types of random disk files. With the "relative byte address" option, random files of variable length records can be processed. Disk file data can be written in either ASCII format or internal binary format.

The CALL builtin procedure provides direct access to all CP/M operating system services. The MAP builtin procedure allows any region of main storage to be accessed as if it were a Pascal variable. Hardware input/output ports are directly accessible.

Debugging is simplified by the line number trace and the procedure name trace which can both be turned on and off by the program at run-time.

Activan - the activity analyzer - can be used to monitor the execution of a program and print out a histogram showing the amount of activity in each program area.

1.2 Hardware requirements

The compiler requires a minimum of 56K of main storage. One disk drive with at least 90K of storage is needed but two or more are strongly recommended.

1.3 List of files

JRT Pascal compiler

```
JRTPAS3.COM
PASCAL0.INT
PASCAL1.INT
PASCAL2.INT
PASCAL3.INT
PASCAL4.INT
PASCAL.LIB
```

Run-time environment

```
EXEC.COM
```

External functions

```
ARCTAN.INT
COS.INT
EXP.INT
LN.INT
SIN.INT
SQRT.INT
```

External procedure assembler

```
JRTASM.INT
```

External procedure linker

```
LINKER.INT
```

CRT Mapping utility

```
CRTMAP.PAS
```

System customization program

```
CUSTOMIZ.INT
```

Block letters external procedure

LETTERS.INT

Indexed file processing procedures
INDEXØ.INT
INDEX1.INT
INDEX2.INT

Table search procedure
SEARCH.INT

Report number formatting facility
PICTURE.INT

Dynamic trace control external procedure
DEBUG.INT

Utility to convert Microsoft modules
CONVERTM.INT

Statistics external procedure
JSTAT.PAS

Graph preparation external procedure
JGRAF.PAS

Sample assembly language external procedures
SETBIT.ASM
RESETBIT.ASM
TESTBIT.ASM

1.4 For Beginners

This section explains how to use JRT Pascal for those who are new to personal computing or who are unfamiliar with "compiled" languages.

This is a tutorial on how to operate our implementation of the Pascal language. For tutorial information on the Pascal language itself, we refer you to the many text books now available. The one book we strongly recommend is the standard definition of Pascal written by its inventor Nicklaus Wirth.

Pascal User Manual and Report
by Jensen and Wirth
published by Springer-Verlag

Developing Pascal programs

Developing a Pascal program is a three step process:

1. create or modify a Pascal source program with any standard CP/M editor like ED or WORDSTAR

2. compile the Pascal source program into an intermediate program

3. execute the intermediate code - run the program

This process is illustrated in the flowchart on a following page.

File names and file types

In CP/M the names of data files and program files consist of two parts: a filename of up to 8 characters and a filetype of up to 3 characters. These two parts are separated by a period.

```
REPORT.LST
A.PAS
A.INT
STAT.COM
```

The JRT Pascal compiler assumes that the source program has a filetype of PAS. It creates an intermediate program with a filetype of INT.

Editors

Any standard CP/M compatible editor may be used to create or modify programs in JRT Pascal. The demo program listing which follows uses the CP/M line editor ED.COM.

Required files **** IMPORTANT ****

The compiler and run-time system are large and complex programs. To make best use of limited main storage they are divided into modules. These modules must be present on your disks when using the compiler or run-time system. The modules need not all be on the A: disk. They may be on either the A: or B: disk, the Pascal system will automatically locate them.

The compiler requires all these files:

```
JRTPAS3.COM
PASCAL.LIB
PASCAL0.INT
PASCAL1.INT
PASCAL2.INT
PASCAL3.INT
PASCAL4.INT
```

The run-time system (execution) requires these files:

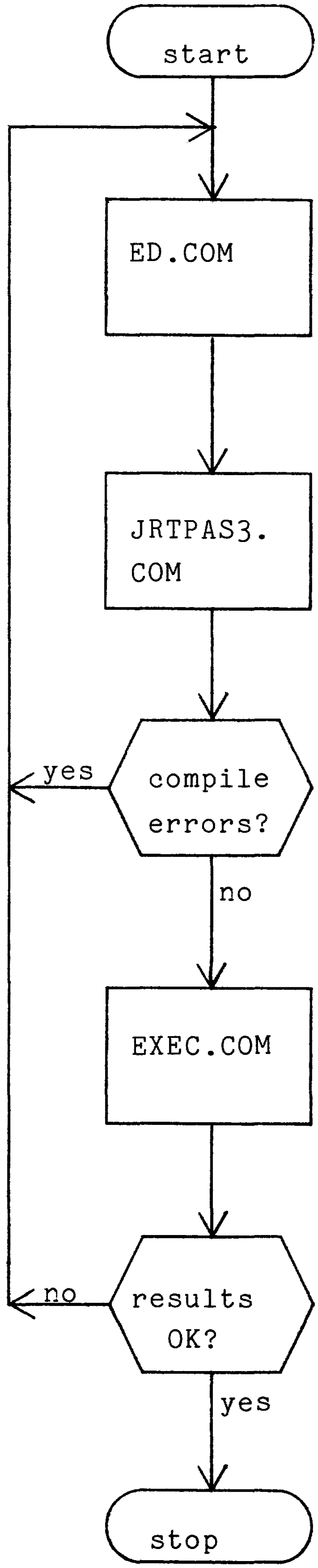
```
EXEC.COM
PASCAL.LIB
```

Demo program

In order to clearly illustrate the program development process, a flowchart of this process is included here. An actual computer listing of the three step process (create, compile, run) for a small demo program follows the flowchart.

The demo program is named A.PAS. It computes and displays the squares of the numbers 1 to 1Ø.

Commands	Actions
ED A.PAS	Create/modify the program
JRTPAS3 A	Compile the progr
EXEC A	Run the program



```
A>ed a.pas
                                  -- Use editor to create program A.PAS
NEW FILE
     : *i
    1:  { demo program to print squares of numbers 1 to 10 }
    2:
    3:  program a;
    4:
    5:  var
    6:  i : integer;
    7:
    8:  begin
    9:  for i := 1 to 10 do
   10:          writeln( i, sqr(i) );
   11:  end.
   12:
     : *e
```

```
jrtpas2 a
                                  -- Compile the demo program
JRT Pascal   ver 2.2

Copyright 1982 JRT Systems

0000  0001:     { demo program to print squares of numbers 1 to 10 }
0000  0002:
0000  0003:     program a;
0000  0004:
0003  0005:     var
0003  0006:     i : integer;
0003  0007:
0006  0008:     begin
0010  0009:     for i := 1 to 10 do
0028  0010:             writeln( i, sqr(i) );
0029  0011:     end.
No errors detected
Module size = 45 dec bytes
End of compile for A
```

```
exec a
Exec  ver 2.2                     -- Run the program

 1 1
 2 4
 3 9
 4 16
 5 25
 6 36
 7 49
 8 64
 9 81
 10 100

Program termination
```

Basic terms

compiler - The Pascal compiler converts Pascal source programs to intermediate program files. It reads in a Pascal source program and writes out an INT file. The compiler also displays the program at the terminal during the compilation process.

debugging - Correcting errors in the program. There are two main categories of errors or "bugs": those which can be detected by the compiler and those which appear only during the execution of the program. Both may be corrected by modifyi the source program and re-compiling.

execution - This is the actual "running" of the program. The run-time environment, EXEC, reads in an intermediate program file from disk and executes its internal computer codes.

intermediate program - This is an internal code version of the program which is created by the compiler. It is a file with a filetype of INT.

source program - This is the actual Pascal program which is a text file and may be printed or viewed on a terminal. It has a filetype of PAS.

trace - There is a JRT Pascal feature which displays the line number of each line in the source program during execution. This is very useful in locating the cause of some program errors.

2. Operating JRT Pascal

JRT Pascal is a fully CP/M compatible language system. The distribution disk does not contain a copy of the operating system due to copyright restrictions. It is recommended that the distribution disk be backed up immediately and not be used as the main running disk.

2.1 Writing Pascal programs

Pascal programs can be developed using any standard editor program. The ASCII character set is used throughout JRT Pascal.

The program file must have a CP/M filetype of 'PAS'. The output modules produced by the compiler, linker and assembler are given a filetype of 'INT'. When the compiler is processing, it creates temporary storage files with a filetype of '$$$'. These are normally deleted but if processing should be interrupted, they may remain on the disk but will be deleted during the next operation of the compiler.

2.1.1 Identifiers

Identifiers are the names assigned to variables, procedures, etc. They may be up to 64 characters long. All characters are significant. They are internally converted to upper case by the compiler.

Identifiers must begin with an alphabetic character. Following characters may be alpha, numeric, the underline character and the dollar sign.

```
x1                          total_value
DISTANCE                    ADDRESS
compute_and_print_average
compute_and_print_totals
MTD_sales                   INITIALIZE_PROC
percent_markup              arc_cotangent
```

Using meaningful data and procedure names greatly improves the readability of programs and serves as self-documentation.

2.1.2 Numbers

Integers or whole numbers in Pascal occupy two bytes of storage and range from -32768 to +32767. In both the Pascal program and in input/output, they can be entered in decimal or hexadecimal format.

Hex format integers have an 'H' suffix character. If the first hex digit is A,B,C,D,E,F then it must be preceded by a zero digit.

```
3AH          ØEADH
12FH         ØcfØØh
-Øffffh      +5Øh
```

Real numbers in JRT Pascal provide 14 digits of precision and floating point capability. The exponent can range from -64 to +63. The numbers are stored in an 8 byte binary-coded-decimal format which eliminates errors in converting between internal and printable formats.

```
3.14159          Ø.ØØØØ98
25ØØØØ.ØØØ321    Ø.442e+35
2.ØE-6Ø          -15.Ø11e+Ø3
```

Real numbers must include the decimal point. The exponent field is optional, but when used must be in a fixed format - character 'e', sign, 2 digits.

2.1.3 Comments

Comments in Pascal can be inserted anywhere in the program. They can be enclosed by either braces { } or by the character pairs (* *).

```
{ comment sample }
(* comment sample # 2 *)
```

2.2 Compiling Pascal programs

JRT Pascal is a one-step compiler, no assembly or link is ever required. The assembler and linker provided are for advanced programming with external procedures.

To compile a program enter:

```
JRTPAS3  filename  <$ options>
```

Examples:

```
JRTPAS3 TESTPGM

JRTPAS3 STATISTC $E

JRTPAS3 INVENTRY $ELP

C:JRTPAS3  B:PROJECT1 $E

JRTPAS3 D:PLOT $E
```

The filetype of the program must be 'PAS'. The filename may be different from the program name.

The compiler option switches are:

E - error stop, interrupt processing on detection of an error, issue message to console, ask user whether or not to continue compiling

L - prepare program for line trace, identical to inserting %LTRACE directive at start of program

P - prepare program for procedure trace, identical to inserting %PTRACE directive at start of program

Tx - control the output listing, x may be:

```
        A..P - write listing to '.LST' file on disk x
        X    - write listing to console device
        Y    - write listing to list device
        Z    - suppress the output listing
```

If errors are detected, verbal error messages will be displayed at the console imbedded in the source listing.

The following files are required by the compiler:

```
JRTPAS3.COM
PASCAL.LIB
PASCALØ.INT
PASCAL1.INT
PASCAL2.INT
PASCAL3.INT
PASCAL4.INT
```

The compiler does not need to be located on the A: disk. The main compiler module JRTPAS3.COM and its external procedures can be placed on any disk drive. Initially, the compiler assumes a two disk system. The CUSTOMIZ program should be used to update the compiler's and EXEC's disk search lists.

2.3 Executing Pascal programs

A program which has compiled with no errors can be executed by entering:

```
EXEC filename <$ options>
```

Examples:

```
B:EXEC D:PLOT

EXEC TESTPGM  $A

EXEC B:PROJECT1
```

The file PASCAL.LIB must be present on one of the disks.

The run-time option switches are:

A - generate an Activan interrupt before program begins execution (refer to appendix for description of Activan)

L - activate the line trace (program must have been compiled with $L option or the %LTRACE directive)

N - generate an Exec interrupt before program begins execution, used for trace control (refer to section on debugging)

P - activate the procedure trace (program must have been compiled with the $P option or the %PTRACE directive)

While the program is running, keying control-a or control-n will cause an Activan or Exec interrupt. At that time certain system parameters can be modified. When in interrupt mode, keying a space character will cause a list of available commands to be displayed. Keying a control-p in interrupt mode causes most system displays to be echoed to the system printer.

If any error or warning conditions occur during the running of the program, a verbal error message is displayed at the console. If the error is severe and the program must terminate, a formatted display of critical system data is provided. This display is described in the section on debugging.

3. Compiler Directives

Compiler directives are instructions to the compiler which are inserted in the Pascal source program. They may be inserted in the program anywhere a comment may appear. (Unlike JRT Pascal version 1, they must not be followed by a semicolon delimiter.)

3.1 Listing Control Directives

When a Pascal program is being compiled, the listing will be displayed on the system console. Three directives are provided to control the program listing.

```
%NOLIST      stop display of program listing

%LIST        resume display of program listing

%PAGE        start a new page in the compiler listing, and
%PAGE(n)       optionally set the "lines per page" value to n

%TITLE('string')    print title at top of each page, activated
             by first %PAGE directive
```

3.2 Line Trace Directives

JRT Pascal line tracing will optionally display the source program line numbers as the program executes. The size of the output module will be increased by three bytes per line.

```
%LTRACE      generate line trace codes
%NOLTRACE    stop generating line trace codes - this
             allows storage saving by tracing only
             a portion of the program
```

JRT Pascal line tracing can be turned on or off under program control by using the SYSTEM builtin procedure. The range of line numbers to be traced can also be modified at run-time by this procedure. WHEN THE PROGRAM BEGINS

EXECUTION, THE LINE TRACE IS DISABLED.

```
    SYSTEM( LTRACE )    activate line trace
    SYSTEM( NOLTRACE )  disable line trace
    SYSTEM( LRANGE, lower, upper )
                        set range of line numbers for
                        line trace - lower and upper are
                        are integer expressions
```

When a program is compiled with the %LTRACE directive, then if the run-time system detects an error condition, the line number will be displayed with the error message.

3.3 Procedure Trace Directives

When procedure tracing is activated, the name of each procedure or function will be displayed on entry and exit. On entry to a procedure the activation count (total number of times called) for that procedure is also listed.

```
    %PTRACE     generate procedure trace codes
    %NOPTRACE   stop generating procedure trace codes
```

Procedure tracing can be turned on or off under program control by using the SYSTEM builtin procedure. WHEN THE PROGRAM BEGINS EXECUTION, THE PROCEDURE TRACE IS DISABLED.

```
    SYSTEM( PTRACE )    activate procedure trace
    SYSTEM( NOPTRACE )  disable procedure trace
```

When a program is compiled with the %PTRACE directive, then if the run-time system detects an error, the name of the procedure most recently activated will be displayed with the error message. Note that the procedure most recently activated is not necessarily the currently active procedure.

If the procedure being entered is an external procedure then the trace message is flagged with an asterisk.

3.4 Source file Include directive

A section of source program code is sometimes used by different main programs or external procedures. Rather than enter this common code at each point it is used, it is easier to use a %INCLUDE directive. This has the effect of inserting the named Pascal code file in place of the directive.

```
%INCLUDE('filename.type')
```

%INCLUDE files may not be nested. This directive should be placed on a line by itself. If the %INCLUDE is indented with spaces then the entire included file is also indented by the same amount.

```
%INCLUDE('GLOBALS.LIB')
%INCLUDE('C:VARDCLS.PAS')
     %INCLUDE('B:SORTPROC.OLD')
```

4. Data types

Pascal is a language rich in data types. Unlike Basic which provides only two or three data types, Pascal provides eight - integers, real numbers, Booleans, characters, structured variables, sets, pointers and dynamic strings. These forms can be combined in records and arrays to form data aggregates that closely relate to the application area. Records and arrays can contain other records and arrays and pointers with no restrictions on nesting or even on recursive definitions.

It is these features that set Pascal apart from earlier languages like Cobol, Fortran, PL/I. Pascal recognizes the importance of powerful facilities for describing the data in a program as well as the active statements.

4.1 Integers

Integers or whole numbers occupy two bytes. They are represented in twos complement format. The range is -32768 to +32767.

Integer literals in the source program and in console or disk input may be entered as hex values. Standard Intel hex format is used. The last character must be an 'H'. A leading zero is required if the first digit is A, B, C, D, E, F.

```
1ah   +0C35H   -0ffh   0c000h   1234H
```

4.2 Real numbers

Real numbers have 14 digits and are expressed in floating point format. The exponent range is from -64 to +63. The exponent field is not required in source program or input but when present must be entered in a fixed format. The exponent format is 'e+00' or 'e-00'.

32.Ø1e+Ø4 1.Ø75 -3.14159 -1234567.89Ø1234E-47

In source programs the decimal point must be included to distinguish real numbers from integers.

4.3 Booleans

Boolean variables may have only two values - TRUE or FALSE. Booleans may be used directly in output statements but should not be used directly in input statements.

4.4 Char

The char data type is one character. Packed char fields are not meaningful on 8-bit microcomputers and are not supported. The ASCII character set is used in JRT Pascal.

4.5 Structured variables

Structured variables are records or arrays which are treated as aggregates. For example - a record of one type could be compared directly against a record of another type. Structured variables may be compared (all six operators), assigned, input/output, concatenated, used as parameters and function return values without restriction.

In addition to the CONCAT builtin function, the '+' operator indicates concatenation of structured variables or dynamic strings.

Structured variables to be compared may have different lengths. The result is determined as if the shorter one were extended by spaces.

In assigning structured variables of different lengths if the receiving field is shorter, truncation occurs. If the receiving field is longer then the remainder of it is padded with spaces.

Arrays of type char constitute fixed length strings. Unlike dynamic strings, these have no (hidden) two byte length prefix. Arrays of fixed length strings are useful for many types of text processing.

```
TYPE
CHAR100 = ARRAY [1..100] OF CHAR;
TABLE = ARRAY [1..40] OF CHAR100;
VAR
T : TABLE;
BEGIN
T := ' ';        (* CLEARS ENTIRE TABLE *)
T[1][8] := '*'; (* STORE 1 CHARACTER   *)
T[15] := 'JRT Pascal is the best';
...
END;
```

4.6 Dynamic strings

Dynamic strings are an extension to standard Pascal. A hidden two byte prefix on the string contains the string's current length in bytes. JRT Pascal dynamic strings may be up to 64K bytes in length - of course the computer's main storage size restricts the size to a smaller value. Other Pascals limit strings to 255 bytes.

The maximum size of a string variable is declared with the variable definition. If no size is specified the default is 80 bytes.

```
VAR
S1 : STRING;
S2 : STRING[4000];
S3 : STRING[12];
```

Dynamic strings may be used in the same way as structured variables - comparisons, assignment, input/output, parameters, function return values.

NOTE - Dynamic string variables may not be used in READ statements directed to files, only to the console. To read string data from files, fixed strings (arrays of characters) must be used.

The individual characters of a string may be accessed and updated. If an attempt is made to access an element of a string beyond the current length of the string, a run-time error occurs.

```
S1[4] := 'X';
WRITELN( S2[15ØØ] );
S1[J] := S1[J+1];
S3[1] := UPCASE( S3[1] );
```

Several builtin procedures and functions are available to enhance string processing. Refer to the sections on builtin functions and on builtin procedures for complete descriptions.

name	purpose
CONCAT	concatenate n strings
COPY	extract portion of string
DELETE	delete portion of string
INSERT	insert a string into another
LENGTH	return current string size
POS	search string for a pattern

4.7 Sets

Set variables occupy 16 bytes. The entire ASCII character set may be represented in the 128 bits.

```
LOW_CASE := ['a'..'z'];
UP_CASE  := ['A'..'Z'];
NUMERIC  := ['Ø'..'9'];
ALPHAMERIC := LOW_CASE + UP_CASE + NUMERIC;
ALPHABETIC := ALPHAMERIC - NUMERIC;

IF NOT (INPUT_CHAR IN ALPHAMERIC) THEN
        WRITELN('INVALID INPUT CHAR');
```

NOTE - Set variables have no meaningful format in text format input/output. Sets may be input/output to disk files which are opened for binary format processing.

4.8 Pointers

Pointers contain the virtual address of dynamic variables created by the NEW procedure and of ghost variables created by the MAP procedure. Pointers are two bytes in size.

The value stored in a pointer variable is NOT the actual address of the dynamic variable - it is the virtual address. The actual address of a dynamic variable may be obtained with the ADDR builtin function.

```
ACTUAL_ADDRESS := ADDR( PTR^ );
```

Note that the actual address of a dynamic variable may change during program execution but the virtual address is fixed for the life of the variable.

4.9 Dynamic arrays

Dynamic arrays are a JRT extension to the Pascal language. Arrays are a widely used device for storing and retrieving logically identical data elements.

Often it is not known in advance how many data elements will be processed - thus it is necessary to create arrays to hold the maximum number of elements that ever may be processed.

With dynamic arrays, the array's actual size need not be "hard-coded" into the source program. The array size may vary with each run of the program or even at different times within the same run.

In some programs, dynamic arrays can greatly improve storage use efficiency. This implies that the program can operate over a much wider range of situations.

IMPORTANT - Dynamic arrays MUST be actual variables - they may NOT be elements of other arrays or fields of record variables. Files of dynamic arrays are not allowed.

Declaring dynamic arrays

The declaration of dynamic arrays in either the TYPE or VAR sections is identical to static arrays except that the indexes are not specified as subranges. The indexes must be specified as either the reserved word INTEGER or CHAR. No other index declaration is allowed in dynamic arrays. Static and dynamic indexes may not be mixed in the same array declaration.

```
TYPE
MATRIX = ARRAY [ INTEGER, INTEGER ] OF REAL;

VAR
M : MATRIX;
TABLE : ARRAY [ CHAR ] OF STRING [20];
INDEX : ARRAY [ INTEGER, CHAR ] OF INTEGER;
```

Allocating and deallocating dynamic arrays

A dynamic array may not be referenced until it has been allocated. Doing so would cause a run-time error. Allocation accomplishes two purposes:

1. establish the dynamic arrays current lower and upper index bounds for each dimension

2. allocate storage for the dynamic array in dynamic storag

Current bounds are stored in an array control block (ACB) which also contains an allocation flag, dimension count, and the virtual address of the dynamic array.

A builtin procedure performs the allocation operation.

```
ALLOCATE ( dyn_array_variable [ subrange_expr1,...
```

```
                              subrange_expr_n ] );
```

Note that an ALLOCATE must be used for each array VARIABLE declared, NOT for array TYPEs.

```
ALLOCATE ( M [1..1Ø, Ø..5Ø] );
ALLOCATE ( TABLE ['A'..'M'] );
ALLOCATE ( INDEX [I..I+1Ø, CHAR1..CHAR2] );
```

The bounds of a dynamic array may be changed by executing another ALLOCATE with different parameters. The data stored in a dynamic array is lost when it is re-ALLOCATEd.

Dynamic arrays follow the standard Pascal rules for scope of reference. They remain allocated until they are explicitly deallocated.

Since dynamic arrays use storage, they should be deallocated when they are no longer needed.

```
DEALLOCATE ( dyn_array_variable );

DEALLOCATE ( M );
DEALLOCATE ( TABLE );
DEALLOCATE ( INDEX );
```

Dynamic arrays declared and allocated within a procedure are not automatically de-allocated on the termination of that procedure.

5. Builtin functions

JRT Pascal provides numerous builtin functions and several external functions. JRT extensions are indicated with an asterisk. External functions are marked with an 'x'.

	function	return value
	ABS	absolute value, integer/real
*	ADDR	address of variable
x	ARCTAN	arc tangent
	CHR	convert integer to character
*	CONCAT	concatenate n strings
*	COPY	extract portion of string
x	COS	cosine
x	EXP	exponential
*	FREE	amount of free space
*	HEX$	convert variable to hex format
*	LENGTH	length of string
x	LN	natural logarithm
	ODD	test for odd value
	ORD	convert character to integer
*	PORTIN	hardware port input
*	POS	search string for pattern
	PRED	preceding value
*	REAL$	convert real number to string
	ROUND	convert real number to integer
x*	SEARCH	fast table search
x	SIN	sine
	SQR	square, integer/real
x	SQRT	square root
	SUCC	succeeding value
	TRUNC	convert real number to integer
*	UPCASE	convert string to upper case

5.1 ABS

Format 1
ABS(integer_expression);

Format 2
ABS(real_expression);

The ABS standard function returns the absolute value of an integer or a real expression.

Examples:

```
A := ABS( X );

WRITELN( 'ABSOLUTE VALUE IS',ABS( COS( Y )));

B := ABS( X + Y / Z );
```

5.2 ADDR

Format
ADDR(variable);

The ADDR function returns the real address of any variable, array element, field of a record, dynamic variable.

Note that the address of a dynamic variable may change when a storage compression occurs. If the address of a dynamic variable is needed, the ADDR function should be used to obtain the current address immediately before use.

Examples:

```
ADDRESS_OF_X := ADDR( X );

AD := ADDR( MATRIX[ X, Y+5 ]);

DYN_VAR := ADDR( BASE^ );

DYN_VAR_2 := ADDR( BASE^.NEXT^ );
```

5.3 ARCTAN

Format
ARCTAN(real_expression);

This standard function returns the arc tangent of a real expression.

This is implemented as an external function. The declaration for an external function must be included in programs which reference it.

```
FUNCTION ARCTAN ( X : REAL ): REAL; EXTERN;
```

Examples:

```
WRITELN( ARCTAN( A + 3.14159 ));

NODE.VALUE := OLD_NODE.VALUE + ARCTAN( V );
```

5.4 CHR

Format

```
CHR( integer_expression );
```

The CHR standard function converts an integer expression into a character. It is often used in sending control characters to output devices.

Examples:

```
WRITE( CHR( 12 ));

WHILE PORTIN( MODEM ) = CHR(ØFFH) DO  I:=I+1;

TAB := CHR( 9 );

CARRIAGE_RETURN := CHR( ØDH );

LINE_FEED := CHR( ØAH );
```

5.5 CONCAT

Format

```
CONCAT( stringexpr1, stringexpr2,..., stringexprn );
```

The CONCAT string function concatenates two or more dynamic strings, literal strings or structured variables. It returns a value of dynamic string of the length required.

The plus sign can also be used to concatenate string expressions.

Examples:

```
OUTPUT_LINE := CONCAT( NAME, TAB, TAB, PHONE );

WRITELN( CONCAT( 'VALUE', OPER, VALUE );

WRITELN( 'VALUE' + OPER + VALUE );
```

5.6 COPY

Format
COPY(string_expression, position, length);

The COPY function returns a string value extracted from the source_string beginning at position for length characters. The position and length parameters are integer expressions. The first character of strings is at position 1. An error will occur if an attempt is made to copy from an area greater than the length of the string.

Examples:

```
CH := COPY( 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                CH_NUM, 1 );

WRITELN( COPY( STR, POS( STR,'*' ), 5 );

WRITELN( COPY( 'THIS IS A STRING', 6, 4 );
(* OUTPUT OF ABOVE LINE IS 'IS A' *)
```

5.7 COS

Format
COS(real_expression);

The COS standard function returns the cosine of a real expression.

This is implemented as an external function. The declaration for an external function must be included in programs which reference it.

```
FUNCTION COS ( X : REAL ): REAL; EXTERN;
```

Examples:

```
WRITELN( COS( ANGLE ));

NODE.COSINE := COS( N );

WRITELN( COS( VELOCITY / CHARGE ));
```

5.8 EXP

Format
EXP(real_expression);

The EXP function computes e to the x power, where x is a real_expression.

This is implemented as an external function. The declaration for an external function must be included in programs which reference it.

```
FUNCTION EXP ( X : REAL ): REAL; EXTERN;
```

Examples:

```
X := EXP( Y );

PROJECTED_SALES := 1000 * EXP( YEAR / 100 );

VOLTAGE := EXP(  SIN( PHASE ) );

SHIP_VELOCITY := EXP( WARP_FACTOR );
```

5.9 FREE

Format
FREE

The FREE integer function returns the amount of storage currently available. Because the virtual storage manager may delete inactive external procedures, much more storage may be potentially available. The FREE function returns a 16-bit integer value.

If more than 32K of storage is available, the value of the integer would print out as negative, due to the limit on integer size. The following function converts unsigned integers to real number format to provide positive representation for numbers up to 65535.

```
FUNCTION REALFREE : REAL;
VAR
TEMP : INTEGER;
BEGIN
TEMP := FREE;
IF TEMP >= Ø THEN
        REALFREE := TEMP
ELSE
        REALFREE := 65536.Ø + TEMP;
END;
```

Examples:

```
WRITELN('FREE SPACE =',FREE);

IF REALFREE <= 2ØØØ.Ø THEN
        WRITELN('STORAGE CRITICAL');

IF FREE >= 15ØØ THEN NEW( BUFFER );

IF FREE >= 4Ø96 THEN BUFSIZE:=2Ø48
ELSE BUFSIZE:=1Ø24;
RESET( INFILE, 'TEST.DAT', BINARY, BUFSIZE );
```

5.1Ø HEX$

Format
HEX$(any_variable);

The HEX$ function converts any variable to hex format for display. The result is of type string and its length is twice the length in bytes of the input variable.

Note that the 8Ø8Ø/Z8Ø microcomputers represent 16 bit integers in byte-reverse format, with low order byte followed by high order byte. That is, +ABCDH would appear in storage as CDAB. The HEX$ function converts all variables as they appear in storage. Often it is useful to display hex integers in the more usual order ABCD. The HEXINT function below makes this conversion.

```
FUNCTION HEXINT ( X : INTEGER ): STRING[4];
VAR
A : STRING[4];
BEGIN
A := HEX$(X);
HEXINT:='    ';
HEXINT[1]:=A[3];
HEXINT[2]:=A[4];
HEXINT[3]:=A[1];
HEXINT[4]:=A[2];
END;
```

Examples:

```
WRITELN( HEX$( 3.14159 ));

WRITELN( HEXINT( ADDR( PTR^ )));

WRITELN( HEXINT( ADDR( FCB )));
```

5.11 LENGTH

Format
LENGTH(dynamic_string_variable);

The LENGTH function returns an integer value which is the current length of the string variable.

IMPORTANT - LENGTH may only be used with dynamic string variables, not with expressions or any other data type.

Examples:

```
        WRITELN( LENGTH( STR1 ) );

        IF LENGTH(STR1) < 75 THEN
                STR1:=CONCAT( STR1, '----' );

        FOR I:=1 TO LENGTH( NAME ) DO
                IF NOT (NAME[I] IN ALPHAMERIC) THEN
                        NAME[I]:=' ';
```

5.12 LN

Format
LN(real_expression);

The LN function computes the natural logarithm of a real expression.

This is implemented as an external function. The declaration for an external function must be included in programs which reference it.

```
FUNCTION LN ( X : REAL ): REAL; EXTERN;
```

Examples:

```
X := LN( Y );

WRITELN( LN( X + SQR(Y)));

IF LN( ATOM_WEIGHT ) < 1000.0 THEN
        WRITELN(F1; ATOM);

A := SQRT( LN(Z));
```

5.13 ODD

Format
ODD(integer_expression);

ODD is a Boolean function which returns the value true if the integer_expression is odd otherwise it returns false.

Examples:

```
IF ODD(X) THEN  TEST_FOR_PRIME(X);

IF ODD(I) THEN I:=I+1;

WHILE ODD( PORTIN(15H)) DO X:=X+1.Ø;

WRITELN( ODD(Y) );
```

5.14 ORD

Format
ORD(character_expression);

The ORD function converts a character to an integer value. The character_expression may be a single character or a string. If it is a string, then the first byte will be converted to integer format. The conversion is based on the ASCII character set.

Example:

```
REPEAT
  READ(INFILE; CH)
  WRITE( CH );
UNTIL ORD(CH) = 1AH;      (* EOF *)

(* ASCII DISPLAY *)
FOR CH := ' ' TO 'z' DO
        WRITELN( CH, ' = ',ORD(CH));

X := ORD( COPY( S1, I, 1 ));
```

5.15 PORTIN

Format
PORTIN(integer_expression);

The PORTIN function inputs a byte directly from the hardware port specified by the integer expression. The return value is a character.

Examples:

```
IF PORTIN(255) = CHR(80H) THEN
        WRITELN('HIGH BIT IS ON');

CH := PORTIN(TTY);

WHILE PORTIN(MODEM) = CHR(0FFH) DO
        TIMER := TIMER + 1.0;
```

5.16 POS

Format 1
POS(pattern, source);

Format 2
POS(pattern, source, start_position);

Search the source string for the first occurence of the pattern string. Return the position of the first byte of the pattern if it was found, otherwise return zero. The first byte is position 1.

In format 2 of the POS function, the start position of the search in the source string can be specified.

```
PROGRAM DEMO;
VAR
  STR1,STR2 : STRING;
BEGIN
STR1 := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
WRITELN( 'TEST 1 :', POS('EF', STR1));
WRITELN( 'TEST 2 :', POS('D', STR1, 8));
STR2 := 'XX  XX  XX';
WRITELN( 'TEST 3 :', POS(' ', STR2));
WRITELN( 'TEST 4 :', POS('XX', STR2, 2));
END.
```

```
OUTPUT:
TEST 1 : 5
TEST 2 : Ø
TEST 3 : 3
TEST 4 : 5
```

5.17 PRED

Format 1

```
PRED( integer_expression );
```

Format 2

```
PRED( character_expression );
```

The PRED function returns preceding value of an integer or a character expression. For example, the PRED of 'c' is 'b', the PRED of 98 is 97.

Example:

```
WRITELN( A, PRED(A) );

WRITELN( CH, PRED(CH) );
```

5.18 REAL$

Format
```
REAL$( real_expression );
```

The REAL$ function converts a real_expression to a printable standard format for direct output or further editing. The output is a string of length 22, in the format below:

```
' +Ø.12345678901234E+ØØ'
```

Examples:

```
WRITELN( FREQUENCY_FILE;
         REAL$( CYCLES / MICROSECONDS ));

STR := REAL$( VELOCITY / 7.Ø3E-21 );
```

5.19 ROUND

Format
ROUND(real_expression);

ROUND is a standard function which converts a real expression to an integer value. If the real value's fractional part is greater than or equal to Ø.5 then the value is rounded up to the next higher integer.

If the real value is too large to be converted to integer format, a warning message is issued and the value returned is -32768 if the real expression was negative otherwise +32767.

Examples:

```
INT := ROUND( X + Y );

TEMPERATURE := ROUND( THERMOMETER_READING );

PLOT_X := ROUND( X / SCALING_FACTOR );
```

5.2Ø SEARCH

Search is an external function which allows high speed searches of tables. The array of records to be searched can be any length, the individual records can be any length, the offset to the key within the record can be specified, and the key length can be specified.

Search takes four arguments: the array, the key, the number of records in the array, and the search parameter record. The count of records in the array is passed by value. The three other arguments are passed by reference.

Declarations required to use SEARCH

```
          TYPE
               search_param = RECORD
                    search_mode : integer;     (* must be
zero *)
                    record_length : integer;
                    key_offset : integer;
                    key_length : integer;
                    END;
               record_type = RECORD
                    (* whatever is appropriate *)
                    END;
                 record_array    =    ARRAY[1..whatever]    OF
record_type;
               key_type = STRING or ARRAY[1..x] OF CHAR;

          VAR
               arr : record_array;
               key : key_type;
               parameters : search_param;

          FUNCTION SEARCH ( VAR arr : record_array ;
                            VAR key : key_type ;
                            count : INTEGER;
                            VAR param  :  search_param  )  ;
EXTERN;
```

Using SEARCH

Set up the search parameter block (generally just once):

```
parameters.search_mode := Ø;
parameters.record_length := (* whatever *);
parameters.offset := Ø (* or whatever *);
parameters.key_length := (* whatever *);
```

SEARCH looks through an array of records for an exact match between the search key and the key within the records. The search_mode option is provided for future extensions to allow the array to be in sorted order, to return the closest record, to let the array to be searched be a linked list, or for the record to contain a pointer to the key.

SEARCH returns -1 if the arguments are invalid, Ø if the key cannot be found, and the index of the record if the key can be found (starting at 1).

Example

For example, assume an array of records containing an integer index and a 6 character key.

```
(* type declaration *)
search_param = RECORD
     search_mode :  integer;     (*  must  be
zero *)
     record_length : integer;
     key_offset : integer;
     key_length : integer;
     END;
char6 = ARRAY[1..6] OF CHAR;
record_type = RECORD
     index_val : INTEGER;
     key : char6;
record_array = ARRAY[1..999] OF record_type;
key_type = char6;

(* variables *)
arr : record_array;
key : key_type;
parameters : search_param;
nr_records : INTEGER;    (* number of records
*)
```

```
          FUNCTION SEARCH ( VAR arr : record_array ;
                            VAR key : key_type ;
                            count : INTEGER;
                            VAR param  :  search_param  )  ;
EXTERN;

          (* setup *)
          parameter.mode := Ø;
          parameter.record_length := 8;
          parameter.key_offset := 2;
          parameter.key_length := 6;

          (* build an array of keys and indices into arr *)
          (* keep track of number of records  in  nr_records
*)

          (* use *)
          ind := search ( arr, key, nr_records, parameter  )
;
          if (ind <= Ø) then
               writeln('Record not found: ', key)
          else
               begin
               (* ... *)
               end;
```

Record lengths and offsets

Record lengths and offsets can be determined by counting bytes. Characters take 1 byte, integers, boolean, and enumerated types take 2 bytes, real numbers take 8 bytes.

5.21 SIN

Format
SIN(real_expression);

The SIN standard function returns the sine of a real expression.

This is implemented as an external function. The declaration for an external function must be included in programs which reference it.

```
FUNCTION SIN ( X : REAL ): REAL; EXTERN;
```

Examples:

```
WRITELN( SIN( ANGLE ));

NODE.SINE := SIN( N );

WRITELN( SIN( VELOCITY / CHARGE ));
```

5.22 SQR

Format 1
SQR(real_expression);

Format 2
SQR(integer_expression);

The SQR standard function returns either a real value or an integer value depending on the parameter type. This function returns the square of the parameter expression - the value multiplied by itself.

Examples:

```
WRITELN( 'SQUARE OF X IS ', SQR(X) );

AREA := SQR( SIDE );

CIRCLE_AREA := PI * SQR( RADIUS );

ENERGY := MASS * SQR( LIGHT_SPEED );
```

5.23 SQRT

Format
SQRT(real_expression);

This standard function returns the square root of a real expression.

This is implemented as an external function. The declaration for an external function must be included in programs which reference it.

```
FUNCTION SQRT ( X : REAL ): REAL; EXTERN;
```

Examples:

```
WRITELN( SQRT( A + 3.14159 ));

NODE.VALUE := OLD_NODE.VALUE + SQRT( V );
```

5.24 SUCC

Format 1
```
SUCC( integer_expression );
```

Format 2
```
SUCC( character_expression );
```

The SUCC function returns succeeding value of an integer or a character expression. For example, the SUCC of 'b' is 'c', the SUCC of 97 is 98.

Example:

```
WRITELN( A, SUCC(A) );

WRITELN( CH, SUCC(CH) );
```

5.25 TRUNC

Format
TRUNC(real_expression);

TRUNC is a standard function which converts a real expression to an integer value. The fractional portion of the real expression is truncated.

If the real value is too large to be converted to integer format, a warning message is issued and the value returned is -32768 if the real expression was negative otherwise +32767.

Examples:

```
INT := TRUNC( X + Y );

TEMPERATURE := TRUNC( THERMOMETER_READING );

PLOT_X := TRUNC( X / SCALING_FACTOR );
```

5.26 UPCASE

Format
UPCASE(string_expression);

The UPCASE function converts a string expression to all upper case letters. Non-alphabetic characters are not changed.

Examples:

```
IF UPCASE( COMMAND ) = 'X' THEN
        CMD_X;

WRITE( F1; UPCASE(NAME) );

READLN( OPTION );
IF UPCASE( OPTION ) = 'EXIT' THEN GOTO 99;
```

6. Builtin procedures

Several builtin procedures are provided in Pascal. Most of these relate to input/output processing and are discussed in the input/output section. The remaining procedures are covered in this section. A list of them and their purpose follows. JRT Pascal extensions are marked with an asterisk.

	procedure	purpose
*	CALL	direct access to CP/M and BIOS
*	DELETE	delete portion of dynamic string
	DISPOSE	de-allocate dynamic variables
*	FILLCHAR	initialize a string
*	INSERT	insert string into dynamic string
*	MAP	access main storage
	NEW	allocate dynamic variables
*	PORTOUT	hardware port output
*	SYSTEM	EXEC services

6.1 CALL

Format
CALL (address, parameter_regs, returned_regs);

The CALL builtin procedure allows you to make direct calls to the CP/M operating system, to your own Basic Input/Output System (BIOS), and to any machine language code present in main storage. The 8080 data registers can be directly setup for passing parameters to the module called. The 8080 data registers which are returned from the module may contain return values which can be used directly from Pascal programs.

Note that this assembly language interface complements the external procedure assembler. User subroutines which must be written in assembler will usually be written as external procedures and assembled. That gives the advantage of fully automatic loading and relocation. CALL is intended primarily for direct access to the operating system services.

The address field is an integer expression. This field is regarded as an unsigned 16-bit integer. When CALL is executed, control is transferred to the machine code at the address. The module there must return control to Pascal with a RET instruction. The 8080 stack pointer must not be modified on return to Pascal.

The 8080, 8085, Z80 microcomputers have 7 one byte data registers and a one byte flag register. The Z80 has additional registers but these are not used in a CP/M environment. Six of the data registers can be grouped as two byte registers for some uses.

8080 Register Map

```
-----------------
I  A    I FLAG  I
-----------------
I  B    I  C    I
-----------------
I  D    I  E    I
-----------------
I  H    I  L    I
-----------------
```

The parameter_regs and returned_regs fields have a particular format which must be declared in your program. The parameter_regs field is directly loaded into the microprocessors data registers before control is transferred to the called module. When control is returned to Pascal, the current data registers are stored into the field identified by returned_regs. Both of these fields should be declared like this:

```
TYPE  DATA_REGISTERS =
        RECORD
        CASE INTEGER OF
        1 : ( FLAG,A,C,B,E,D,L,H : CHAR );
        2 : ( PSW,BC,DE,HL : INTEGER );
        END;
```

This is a variant record which defines the data registers for access in one or two bytes at a time. For example, sometimes it may be necessary to regard the register pair DE as an integer, other times it may be necessary to treat register E alone as a single byte. Both definitions total 8 bytes.

Note that in definition 1, the register names are in an unusual sequence. This is necessary because the 8080/Z80 microprocessors store 16 bit data in a "byte-reverse" format.

Example:

```
VAR
PARM_REGS, RETURNED_REGS : DATA_REGISTERS;

CALL( 5, PARM_REGS, RETURNED_REGS );
```

6.1.1 Calling the CP/M operating system

An operating system is a program which provides services to application programs running under it. Some of these services are "create file", "write string to printer", "reinitialize system", and so on. Using the CALL builtin procedure you can directly access these services from your Pascal programs.

The CP/M and MP/M User's Guides describe in detail the services provided and parameters required for each. Each service is identified by a one byte function code. This code is stored in register C before control is transferred to CP/M. Many services also require an integer parameter such as an address in register pair DE. The entry point address for all CP/M compatible systems is location 5. At address 5 is stored a jump instruction to the actual CP/M module.

The address of the BIOS (warm-start entry point) is stored at address ØØØ1 in main storage and may be accessed with the MAP builtin procedure. The MAP and CALL procedures allow direct access to all of the services provided by the BIOS.

The service codes for CP/M 2.2 and MP/M are:

0	system reset
1	console input
2	console output
3	reader input
4	punch output
5	printer output
6	direct console input/output
7	get I/O byte
8	set I/O byte
9	print string
10	read console buffer
11	get console status
12	return version number
13	reset disk system
14	select disk
15	open existing file
16	close file
17	search for first file control block
18	search for next file control block
19	delete file
20	read sequential
21	write sequential
22	create file
23	rename file
24	return login vector
25	return current disk
26	set DMA address
27	get addr (alloc)
28	write protect disk
29	get read/only vector
30	set file attributes
31	get addr (disk parms)
32	set/get user code
33	read random record
34	write random record
35	compute file size
36	set random record
37	reset drive
40	write random with zero fill

The following services are available in MP/M only:

128	absolute memory request
129	relocatable memory request
13Ø	memory free
131	poll
132	flag wait
133	flag set
134	create queue
135	open queue
136	delete queue
137	read queue
138	conditional read queue
139	write queue
14Ø	conditional write queue
141	delay
142	dispatch
143	terminate process
144	create process
145	set priority
146	attach console
147	detach console
148	set console
149	assign console
15Ø	send CLI command
151	call resident system process
152	parse filename
153	get console number
154	system data address
155	get date and time

Examples:

1.

```
(* GET THE VERSION NUMBER FROM CP/M *)

PROCEDURE GET_VERSION;
VAR
PARM_REGS, RETURN_REGS : DATA_REGISTERS;
BEGIN
(* SET FUNCTION CODE := 12 *)
PARM_REGS.C := CHR(12);
CALL( 5, PARM_REGS, RETURN_REGS );

(* THE CP/M VERSION NUMBER IS RETURNED IN
REGISTER L.  IF REGISTER H IS Ø1 THEN THE
OPERATING SYSTEM IS MP/M                   *)
CASE ORD( RETURNED_REGS.H ) OF
Ø : WRITE('CP/M ');
1 : WRITE('MP/M ');
ELSE : WRITE('????');
END;
WRITE(' VERSION ');

CASE HEX$( RETURNED_REGS.L ) OF
'ØØ' : WRITELN('1.X');
'2Ø' : WRITELN('2.Ø');
'22' : WRITELN('2.2');
ELSE : WRITELN( HEX$( RETURNED_REGS.L ));
END;

END; (* GET_VERSION *)
```

2.

```
PROCEDURE WRITE_PROTECT_CURRENT_DISK;
VAR
PARM_REGS, RETURNED_REGS : DATA_REGISTERS;
BEGIN
PARM_REGS.C := CHR(28);
CALL( 5, PARM_REGS, RETURNED_REGS );
END;
```

3.
```
PROCEDURE GET_USER_CODE;
VAR
PARM_REGS, RETURNED_REGS : DATA_REGISTERS;
BEGIN
PARM_REGS.C := CHR(32);
CALL( 5, PARM_REGS, RETURNED_REGS );
WRITELN('USER_CODE =',ORD( RETURNED_REGS.A ));
END;
```

4.
```
PROCEDURE SEARCH_FOR_FIRST
        ( NAME, TYPE : STRING[8] );
TYPE
FILE_CONTROL_BLOCK =
        RECORD
        DISK : CHAR;
        FILENAME : ARRAY [1..8] OF CHAR;
        FILETYPE : ARRAY [1..3] OF CHAR;
        EXTENT : CHAR;
        S1, S2 : CHAR;
        RECORD_COUNT : CHAR;
        BLOCKS : ARRAY [1..16] OF CHAR;
        CURRENT_RECORD : CHAR;
        RØ, R1, R2 : CHAR;
        END;
VAR
FCB : FILE_CONTROL_BLOCK;
PARM_REGS, RETURNED_REGS : DATA_REGISTERS;

BEGIN
(* SET UP FCB *)
FCB.DISK := CHR(Ø);
FCB.FILENAME := NAME;
FCB.FILETYPE := TYPE;

(* SET UP PARM_REGS *)
PARM_REGS.C := CHR(17);
PARM_REGS.DE := ADDR(FCB);
CALL( 5, PARM_REGS, RETURNED_REGS );

(* TEST RETURN CODE *)
IF RETURNED_REGS.A = CHR(255) THEN
        WRITELN('FILE NOT FOUND');
END;
```

6.2 DELETE

Format
DELETE(string_variable, position, length);

The DELETE builtin procedure is used to delete a number of characters from a dynamic string variable. The first parameter refers to the string variable, NOT a string expression. The second parameter is an integer expression which indicates the first character to be deleted - characters in dynamic strings are numbered from 1. The third parameter is an integer expression which indicates the number of characters to be deleted.

The hidden length field of the dynamic string variable is updated. If the position and length parameters refer to an area beyond the current length of the string, a run-time error occurs.

Examples:

```
DELETE( TARGET_STR, 25, 3 );

DELETE( STR1, POS( 'END', STR1), 3 );

DELETE( STR3, 9, X + 3 );
```

6.3 DISPOSE

Format
DISPOSE(pointer_variable);

The DISPOSE builtin procedure is used to de-allocate dynamic variables. The pointer_variable addresses a dynamic variable in dynamic storage. After execution of the procedure the space released is available for other uses.

JRT Pascal supports true dynamic storage with auto-compression. When blocks are freed up, storage fragmentation tends to occur - that is, small unused blocks tend to accumulate. Because many blocks tend to be small, they cannot be immediately reused for another purpose. When storage becomes short an auto-compression is initiated by the Pascal system. In this process all freed blocks are gathered into the center area of storage and all needed blocks are moved to the top of storage. In this way, storage fragmentation is totally eliminated.

The DISPOSE procedure can be used to de-allocate ghost variables created by the MAP builtin procedure. Although ghost variables use no real storage, they do require a small amount of space in the pointer tables.

Example:

```
PROCEDURE DISPOSE_DEMO;
TYPE
DYN_VAR = ARRAY [1..200] OF CHAR;
VAR
POINTER : ^DYN_VAR;
BEGIN
NEW( POINTER );  (* ALLOCATE A DYNAMIC VAR *)

(* DO SOME PROCESSING WITH THE DYNAMIC VAR *)

DISPOSE( POINTER ); (* FREE UP THE 200 BYTES *)
END;
```

6.4 FILLCHAR

Format

```
FILLCHAR( structured_variable, length, character );
```

The FILLCHAR builtin procedure is a very fast and simple way to initialize a structured variable (array or record) to a character. The length parameter is an integer expression which indicates the number of bytes to be initialized. The entire variable from its first byte up to the length specified is set to the character expression value.

CAUTION - This is a hazardous procedure since the run-time system cannot verify that the initialization by character has not run past the end of the variable and perhaps overlayed other variables or program code.

Examples:

```
FILLCHAR( VECTOR, 160, CHR(0) );

FILLCHAR( PRODUCT_ARRAY, 2500, '*' );
```

6.5 INSERT

Format
INSERT(source_string, target_string_variable, position);

The INSERT builtin procedure inserts the source string expression into the target string variable at the indicated position. The source string may be a literal string or other string expression. The target string MUST be an actual variable. The source string is inserted into the target variable beginning at the character indicated by the integer expression position.

If the combination of parameters would cause the target string to overflow its maximum length or if position is less than 1, a run-time error occurs.

Examples:

```
INSERT( 'ABCD', STR1, 15 );

INSERT( FILENAME, MASK, 1 );

STR1 := 'MERE FACTICITY.';
INSERT( 'TRUTH IS NOT ', STR1, 1 );
```

6.6 MAP

Format
MAP(pointer_variable, address);

The MAP procedure allows the user to access any part of the computer's storage. It uses the facilities of the dynamic storage system and pointer variables to, in effect, overlay a map on any area of storage. This is sometimes called a "dsect" or "ghost variable."

Unlike its close relative, the NEW procedure, MAP does not actually allocate a dynamic storage block. Instead of obtaining a storage block and setting the pointer variable to point to it, it lets you specify the address. The address can be anywhere from Ø to ØFFFFH.

Like the NEW procedure, MAP does require five bytes of pointer table space. When the ghost variable is no longer needed, it can be removed from the table with the DISPOSE procedure.

Examples:

1.
```
(* ACCESS A 24 X 8Ø VIDEO TERMINAL       *)
(* IT IS A MEMORY-MAPPED MODEL WITH ITS *)
(* VIDEO SCREEN BEGINNING AT ØFØØØH     *)

TYPE
SCREEN = ARRAY [1..24, 1..8Ø] OF CHAR;
VAR
CRT : ^SCREEN;
BEGIN
MAP( CRT, ØFØØØH );

(* CLEAR THE SCREEN *)
CRT^ := ' ';

(* WRITE MESSAGE ON TOP LINE OF CRT *)
CRT^[1] := 'MEMORY MAPPED CRT EXAMPLE';
...
END;
```

2.
```
(* OBTAIN THE ADDRESS OF THE USER BIOS.*)
(* JMP INSTRUCTION AT ADDR Ø ADDRESSES *)
(* THE WARM-START ENTRY POINT IN BIOS  *)

FUNCTION BIOS : INTEGER;
VAR
PTR : ^INTEGER;
BEGIN
MAP( PTR, 1 );
BIOS := (PTR^ - 3);   (* START OF BIOS *)
END;
```

3.
```
(* SET THE IOBYTE AT ADDR 3 TO NEW VALUE *)

PROCEDURE SET_IOBYTE ( X : CHAR );
VAR
PTR : ^CHAR;
BEGIN
MAP( PTR, 3 );
PTR^ := X;
DISPOSE( PTR );
END;
```

6.7 NEW

Format 1
```
NEW( pointer_variable );
```

Format 2
```
NEW( pointer_variable, tag1,..., tagn );
```

The NEW procedure allocates new dynamic variables. A block of dynamic storage of the required size is obtained. The block's virtual address, not its actual address is stored in the pointer variable.

Virtual addressing and dynamic storage are fully explained in the section on storage management.

After NEW has been executed, the dynamic variable may be accessed. Dynamic variables remain allocated until specifically de-allocated by the DISPOSE procedure. If a procedure uses NEW to allocate a dynamic variable, that variable remains allocated after the procedure ends.

Format 2 contains 1 to n tag fields. These are the fields specified in the CASE clause of variant records.

Example:

```
(* PROGRAM FRAGMENT TO ALLOCATE A        *)
(* LINKED LIST OF VARIABLE LENGTH.       *)
(* THE ROOT OF THE LIST IS A GLOBAL      *)
(* VARIABLE.  NODES AFTER THE FIRST      *)
(* ARE INSERTED BETWEEN THE ROOT AND     *)
(* THE FIRST NODE.                       *)

TYPE
NODE =  RECORD
        NEXT : INTEGER;
        DATA : STRING[300];
        END;
VAR
ROOT : ^NODE;

PROCEDURE LINKED_LIST ( COUNT : INTEGER );
VAR
I : INTEGER;
TEMP : ^NODE;
BEGIN
(* ALLOCATE FIRST NODE *)
NEW( ROOT );

(* SET END_OF_LIST INDICATOR *)
ROOT^.NEXT := NIL;

(* ALLOCATE LINKED LIST *)
FOR I := 1 TO COUNT DO
        BEGIN
        NEW( TEMP );
        TEMP^.NEXT := ROOT;
        ROOT := TEMP;
        END;
END; (* LINKED_LIST *)
```

6.8 PORTOUT

Format
```
PORTOUT( port_number, byte );
```

The PORTOUT procedure writes a byte directly to one of the hardware output ports. The port_number is an integer expression. The byte is a string or char expression.

Examples:

```
PORTOUT( MODEM, START_CHAR );

PORTOUT( VOICE_SYNTHESIZER, 'A' );

PORTOUT( FIRE_ALARM, RESET );

PORTOUT( TELETYPE, CHR(7) );

PORTOUT( 15H, CHR( 3 + X ));
```

6.9 SYSTEM

Format
SYSTEM(option);

The SYSTEM procedure allows you to control the trace facilities, the routing of console output, dynamic storage compression and warning messages.

The options for SYSTEM are listed, default states of the Pascal system are indicated with an asterisk.

```
  option           purpose
  ------           -------
* CONS             route output to console
  NOCONS           no output to console
  LIST             route output to printer
* NOLIST           no output to printer
* WARNING          display warning messages
  NOWARNING        suppress warning messages
  LTRACE           activate line trace
* NOLTRACE         disable line trace
  LRANGE,l,u       set line range for line trace
  PTRACE           activate procedure trace
* NOPTRACE         disable procedure trace
  INITIALIZE       re-initialize disk system
                      after disk switch
  COMPRESS         compress dynamic storage
```

The LRANGE option requires two additional parameters. The lower and upper line numbers are integer expressions.

Examples:

```
SYSTEM( LIST );

SYSTEM( NOWARNING );

SYSTEM( LRANGE, 250, 300 );

SYSTEM( COMPRESS );
```

7. Input/output

JRT Pascal includes a powerful input/output subsystem which can be used to meet virtually any processing requirement. Four modes of input/output - console, sequential disk, random disk, indexed disk - are provided.

Disk files can be processed in either TEXT mode or in BINARY mode. TEXT mode is most commonly used by BASIC languages. Data is stored in ASCII text readable format. BINARY mode is found on larger mini and mainframe computers. The data is input/output in the binary format used internally by the language. Not only is the data more compact in some cases but it is also of fixed length. For example, an integer in text format could occupy from two bytes to six bytes depending on its value. But in binary format, an integer is always exactly two bytes.

Text mode is sometimes called "stream I/O". Binary mode is sometimes called "record I/O".

Another advantage of binary format is that you can process data files or COM files containing special control characters.

All files in JRT Pascal are "untyped". That is you can read and write data of any format to any file. You can write records of entirely different formats and sizes on the same file.

JRT Pascal also supports direct access to the hardware input/output ports without having to write an assembly language subroutine. The builtin function PORTIN and builtin procedure PORTOUT are described in the sections on builtin functions and procedures.

JRT Pascal version 3 now supports Pascal file variables. Files may now be passed as parameters to procedures, allocated locally in procedures, be used in records or arrays, be used in assignment statements. The Pascal builtin procedures GET and PUT are now supported.

7.1 Console input/output

Console input/output is the usual means for a program to interact with the user. Data values can be displayed at a video terminal or teletype and data can be keyed in in response.

Console input/output always occurs in text rather than binary format. Integers, real numbers, strings, characters, Booleans will be displayed in text format. Set variables have no meaningful text format and cannot be written to the console.

IMPORTANT - Since the console is regarded as a text device, data items are delimited by commas, spaces, tabs and semicolons. To read one character at a time use this function:

```
FUNCTION GET_CHAR : CHAR;
VAR R : RECORD
        FLAG,A,C,B,E,D,L,L : CHAR;
        END;
BEGIN
R.C := CHR(1);
CALL( 5,R,R );
GET_CHAR := R.A
END;
```

Using the HEX$ builtin function any variable can be converted to hex format for direct display. On console input for integers, data may be keyed in standard decimal format or in hex format. An 'H' character suffix indicates hex format.

On input to the console, data items may be separated by spaces, tabs, commas or semicolons. Character or structured variable inputs which contain special characters may be entered in single quotes. The quote character itself may be entered by doubling it.

Sample input lines

```
3.14159,77
03ch,'JRT Systems'
'don''t say you can''t'
6.70234e-25,0.0000003
```

Reading from the console into a dynamic string variable is treated differently. An entire line of text is obtained from the console and moved directly into the string variable. Separator characters and single quotes are ignored. The system will not allow more characters to be keyed in than can fit into the variable. The string variable must be the only variable in the READ's parameter list.

Console output can also be routed to the printer or list device. The SYSTEM procedure is fully described in the section on builtin procedures. Some of its options are:

```
SYSTEM( LIST );          route output to printer
SYSTEM( NOLIST );        do not route to printer
SYSTEM( CONS );          route to console device
SYSTEM( NOCONS );        do not route to console
```

The builtin procedures/functions used in console input/output are:

```
READ, READLN    read data into storage
WRITE, WRITELN  write data to console/printer
EOLN            end of line function
```

7.2 Sequential file processing

Disk files are not inherently sequential or random. Those terms apply to the means of access which may be applied to any disk file.

Sequential file processing is generally faster than random access because input/output can be buffered and because the disk positioning mechanism only needs to move short distances.

JRT Pascal lets the user obtain maximum processing speed by defining the buffer size for sequential files. The buffer is the holding area where disk data is loaded and written. This area is filled or emptied in one burst - one disk access with one head load operation. A very small buffer may cause disk "chattering" during processing because of frequent accesses. A large buffer will result in less frequent but longer disk accesses.

The buffer size is specified as an integer expression in the RESET or REWRITE procedure. It will be rounded up to a multiple of 128. If storage is plentiful, buffers of 4096 or 8192 bytes will improve processing.

The builtin procedures/functions used in sequential disk file processing are:

RESET	open file for input
REWRITE	open file for output
CLOSE	terminate file processing
READ, READLN	read data into storage
WRITE, WRITELN	write data to disk
EOF	end of file function
EOLN	end of line function
ERASE	delete a file
RENAME	rename a file

This sample program reads in a file and dumps it in hex format to the console.

```
PROGRAM DUMP;

TYPE  BLOCK = ARRAY [1..16] OF CHAR;
      NAME = ARRAY [1..14] OF CHAR;

VAR
B : BLOCK;
DUMP_FILE : FILE OF BLOCK;
FILENAME : NAME;

BEGIN
WHILE TRUE DO   (* INFINITE LOOP *)
        BEGIN
        WRITE('enter file name : ');
        READLN( FILENAME );
        RESET( DUMP_FILE, FILENAME,
               BINARY, 4096);
        WHILE NOT EOF( DUMP_FILE ) DO
                BEGIN
                READ( DUMP_FILE; B);
                WRITELN( HEX$(B) );
                END;
        CLOSE( DUMP_FILE );
        WRITELN;
        END;
END.
```

7.3 Random file processing

CP/M version 2.2 or higher is required to use JRT Pascal random file processing.

For many types of processing it is not known in advance in which sequence the records of a file will be needed. A spelling dictionary or online inquiry customer database obviously must use random access files.

In JRT Pascal random access is fully supported. Data can be read and updated by providing the relative record number (RRN) within the file for fixed length records. The first record is at RRN = Ø. For variable length records, the data can be read or updated by providing the relative byte address (RBA). The RBA is the location of the data item within the file - the first byte is at RBA = Ø.

The RBA mode of processing gives much greater flexibility than RRN. If all records had to be the same size, then all must be the size of the largest, resulting in much wasted space and slower access.

JRT Pascal version 2.1 now supports random files up to the CP/M maximum of 8 megabytes. The RBA or RRN value may be an integer or a real expression. Programs written under earlier versions are source code compatible but must be recompiled using the version 2.1 compiler.

The procedures used in random file processing are:

OPEN	open or create random file
CLOSE	terminate file processing
READ	read data into storage
WRITE	transfer data to disk
ERASE	delete a file
RENAME	rename a file

A sample program shows random access to a file containing sales information for the various departments of a retail store. The records are located by department number.

```
PROGRAM INQUIRY;

LABEL 10;

TYPE
DEPT_RECORD = RECORD
        INVENTORY          : REAL;
        MTD_SALES          : REAL;
        YTD_SALES          : REAL;
        DISCOUNT           : REAL;
        END;

VAR
INPUT_AREA : DEPT_RECORD;
DEPT_FILE  : FILE OF DEPT_RECORD;
DEPT       : INTEGER;

BEGIN (* INQUIRY *)
OPEN( DEPT_FILE, 'C:DEPTDATA.RND', BINARY );

REPEAT
  WRITE('Enter dept number : ');
  READLN( DEPT );
  IF DEPT = 999 THEN GOTO 10;  (* EXIT *)
  READ( DEPT_FILE, RRN, DEPT;
        INPUT_AREA );
  WRITELN;
  WRITELN('dept',DEPT,
        '   inv',INPUT_AREA.INVENTORY:9:2,
        '   disc',INPUT_AREA.DISCOUNT:9:2);
  WRITELN('   MTD sales',INPUT_AREA.MTD_SALES:9:2,
        '   YTD sales',INPUT_AREA.YTD_SALES:9:2);
  WRITELN;
10: (* EXIT LABEL *)
UNTIL DEPT = 999;

CLOSE( DEPT_FILE );
END (* INQUIRY *).
```

7.4 Indexed file processing

CP/M version 2.2 or higher is required to use JRT Pascal indexed file processing.

JRT Pascal version 3 now provides full support for indexed files. The index file system is implemented as 2 external procedures so that it occupies no main storage when it is not being used.

Indexed files consist of two separate disk files: the main data file with a filetype of DAT and an index file with a filetype of IXØ.

The indexed file system has 3 components. INDEXØ external procedure performs most of the functions. INDEX1 external procedure compresses the data files and rebalances the indexes. The INDEX2 program is executed by itself and reorganizes the files for more efficient access.

The external procedure INDEXØ performs these operations:

A	add a new record
B	read first record (beginning)
C	close file
D	delete a record
F	flush buffers, close and reopen files
N	new file allocation
O	open file
Q	query whether indexes should be balanced
R	read a record
S	read next record in sequence
U	update a record
W	issue warning messages
Z	turn off warning messages

INDEX1 performs these operations.

J	rebalance the indexes
K	compress data file and balance indexes

Records must all be the same size - from 16 to 2Ø48 bytes. They need not be a multiple of 128 bytes. The maximum number of records depends on the key size:

```
(1024 DIV (KEY_SIZE + 3)) * 256

key size          max records
--------          -----------
   4                 32767  <---   Not more than 32767
   6                 28928         records ever allowed
   8                 23808
  15                 14336
```

The maximum number of records should be set to somewhat less than the maximum theoretical number of records, to prevent the loss of a record when adding to an unbalanced file. Note also that the file of indexes will be 257K when the maximum number of records are entered, so a reasonable (high) estimate should be used for the maximum number of records.

IMPORTANT - No key should contain all zeroes, since a zero key is used to indicate deleted keys and records.

The key must be the first field in each record. The key size may be from 2 to 32 bytes.

A utility program INDEX2 is provided to reorganize the data file and generate new index files.

7.4.1 Index file format

The index file is divided into one primary index and up to 256 secondary indexes. Each index block is 1024 bytes.

The primary index contains 256 4 byte fields. Each of these is the first 4 bytes of the lowest key in a secondary index.

The secondary indexes contain actual key values and 3 byte record locator fields. The number of keys per secondary index is:

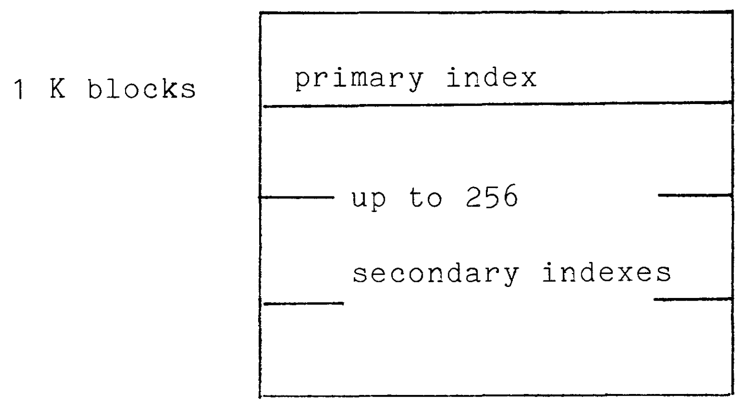
```
1024 DIV (KEY_SIZE + 3)
```

7.4.2 Data file format

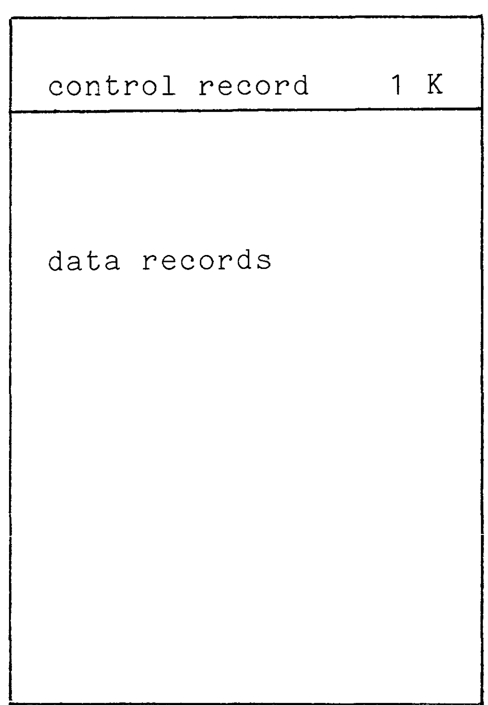
The data file consists of a 1024 byte control record followed by the data records.

The control record contains the filename, maximum record count, current record count, key size, record size, delete count, and deleted record list.

Index file format



Data file format



7.4.3 Using INDEXØ

The indexed file system is implemented in an external procedure named INDEXØ. To access it, these declarations are required in your main program.

```
TYPE
KEY_TYPE =  --------  { your key type declarations }
RECORD_TYPE = ------  { your record type declarations }
INDEX_RECORD = RECORD
        DISK : CHAR;
        FILENAME : ARRAY [1..8] OF CHAR;
        RETURN_CODE : INTEGER;
        RESERVED : ARRAY [1..200] OF CHAR;
        END;

PROCEDURE INDEXØ ( COMMAND : CHAR;
                  VAR KEY : KEY_TYPE;
                  VAR DATA : RECORD_TYPE;
                  VAR IR : INDEX_RECORD ); EXTERN;
```

To use INDEXØ the index_record must be initialized with the filename and disk on which the file is located. The return code is set by INDEXØ and indicates if each operation was successfully completed. Warning messages may optionally be issued, see command 'W'.

An indexed file must be allocated before it can be opened or used in any way.

Each time INDEXØ is called a valid command code must be passed. The key, data, and ir parameters are also required although key and data will not be used by every command.

It is allowed to have multiple indexed files open at the same time. Each one is identified by a different index_record.

The index record (IR) should be set to blanks before individual fields are initialized. For a given index file, the first call to INDEXØ in a program should be to open ('O') or create ('N') the index and data files. (INDEXØ can be called with 'W' first, so that error messages will be printed.)

7.4.4 INDEX commands

Commands J and K are processed by INDEX1. All others are processed by INDEXØ.

A add a new record
- insert new key into index, if duplicate key exists, abort operation
- write new data record to data file

B read first record (begin)
- read the first record (in sorted order)
- returns key and record

C close indexed files
- this MUST be done on completion of processing or newly written data may be lost

D delete a record
- nullify key entry for record
- add record locator to delete list

F flush buffers, close and reopen files
- flush buffers that have changed
- close files to preserve changes

J rebalance indexes (INDEX1)
- uses temporary file
- delets old index file
- renames new index file

K rebalance indexes and compact data file (INDEX1)
- uses temporary files
- deletes old index and data files
- renames new index and data files
- reopen files for further processing

N new file allocation
- program will inquire at the console the parameters of the new indexed file
  1. record size in bytes
  2. key size in bytes
  3. maximum number of records to be allowed; the index file will be allocated based on this number

- index files are left open for further processing
- files must be closed (or flushed) to preserve the new contents

O open indexed files
- open the index and data files
- load the primary index into dynamic storage

Q query data base status
- return 'Y' in key[1] if the data base should be reorganized ('J')
- else return 'N' in key[1]

R read a record
- search the indexes for the key
- read the data record into the user's record variable

S read next record in sequence
- will read next record after a previous 'B', 'R', 'S', or 'U'

U update a record
- the update operation MUST ALWAYS be preceded by a read operation with the same key
- write modified record to data file

W warning messages
- turn on the warning message feature
- causes non-zero return codes to print verbal error messages

Z turn off warning messages

7.4.5 INDEX return codes

Ø	successful completion
1	duplicate key
2	maximum number of records exceeded
3	key not found
4	update key does not match read key or previous read was not successful
5	key value does not match key in record
6	second open or new without closing previous file
7	invalid command (eg. 'M' or an 'S' without a preceeding 'B', 'R', 'S', or 'U')
8	file not open
9	serious error

7.4.6 Balanced indexes

Searching for records is usually very efficient, both in random and sequential modes. Adding to a data base is usually efficient until one or more of the secondary indexes gets full. (If records are added in sorted order, then the addition process will be very efficient.) INDEXØ will not automatically "balance" keys in the index files, so that additions fill up the secondary indexes.

Your program can "Query" the status of an indexed file by using 'Q' in a call the index. The first letter of the key will be set to 'Y' if the indexes should be balanced, and 'N' if that is not necessary yet. (INDEXØ decides that the indexes should be balanced when an add ('A') must move a secondary index from one block to another).

Reorganizing indexes

To reorganize an indexed file so that adding new records will be efficient, set the record argument to all blanks and call INDEX1 with command 'J' (for adJust or Justify). INDEX1 will create a new balanced index file on the same disk as the current index file. There must be space for the new index file, which will be called name.$$I. INDEX1 will then delete the old .IXØ file and rename the new file to name.IXØ. Reorganization takes 25ØØ to 32ØØ bytes of space in main memory as well as space on the disk, so it is never done automatically. INDEX1 must be declared as an external procedure (just as INDEXØ was declared) if your program is going to balance indexes "on the fly".

```
PROCEDURE INDEX1 ( COMMAND : CHAR;
                   VAR KEY : KEY_TYPE;
                   VAR DATA : RECORD_TYPE;
                   VAR IR : INDEX_RECORD ); EXTERN;
```

INDEX1 supports the J and K operations which are described in section 7.4.4.

In general, the record variable should be set to all blanks before INDEX1 is called.

7.4.7 INDEX2 utility

EXEC INDEX2 to rebalance the indexes in the file and to compact the data after many deletions. INDEX2 will ask for the name of the disk drive containing the indexed files (A to P), the name of the index files (which you would enter without any . or .DAT or .IXØ), and the name of the disk drive to contain the new balanced and compacted files. You can have the new files put on the same or another disk drive as the original files.

INDEX2 will also ask for a new number of maximum records. If you enter Ø, the previous maximum will be used.

Compressing data from within a program

INDEX2 uses INDEXØ and INDEX1 to perform the actual indexed file accesses. Highly sophisticated programs can also use INDEX1 to compact the data file as well as balance the indexes. Call INDEX1 with the command 'K' (kompress) to do a complete reorganization. If the record argument is set to all blanks, then the same disk drive and same maximum record count will be used in creating the new data base copies. If the record argument is given the following structure, then alternate disk drives or a different maximum number of records can be set.

```
VAR
     new_param : RECORD
          new_disk_flag: CHAR;
          new_disk : CHAR;
          max_nr_flag : CHAR;
          max_nr_rec : INTEGER;
          old_leave : CHAR;
          END;
```

Set new_param.new_disk_flag to 'Y' if new_param.new_disk contains another disk drive letter (such as 'C'). Set new_param.max_nr_flag to 'Y' if new_param.max_nr_rec contains a new maximum number of records, such as 2ØØØ.

The new_disk_flag only works with the 'K' option. The old_leave flag only works with the 'K' option when a new_disk is specified.

When the 'K' option is used, the record passed must be big enough to hold records read from the disk. You might want to assign rec to contain new_param, and then call INDEX1, for example

```
rec := new_param;
INDEX1 ('K', key, rec, ir);
```

Most programs will not need to use the K option, since the equivalent can be done as needed by having the user issue the CP/M command EXEC INDEX2, preferably after the data bases have been copied to backup disks.

7.4.8 Efficiency notes

Reading records from the data base is only slow when very many keys have the same first four characters. If the indexes in more than one secondary index block have the same first four characters, INDEX0 may have to search more than one secondary index block to find a given record. Generally, this will not occur.

Random output in general under CP/M is inefficient due to buffering requirements. Random output will be most efficient with double density disks with 1k blocks or with single density disks with 128 byte blocks.

Maximum number of records

The maximum number of records should be set to somewhat (50 to 200) less than the theoretical maximum. If, for example, 8 byte keys are declared with up to 23808 records, 256 records are entered, the indexes are balanced (with 'J'). There will now be 256 secondary indexes blocks with one key each. Then, if 92 records are added with key greater than the 256th record, the last secondary index will be full. Since one secondary index block can hold 93 8 byte keys, adding a 93rd key larger than the 256th will "overflow" the top secondary index block. A serious error.

Currently, the maximum number of records is 32767 for index files with 2, 3, and 4 byte keys.

7.4.9 Sample indexed file program

The following simple program will let you create, add to, query, close, and search any data base. It assumes that the record and the key are alphanumeric (printable) information. You can enter individual commands to the program, which will call INDEXØ (or INDEX1) to perform the equivalent command. The runtime example that follows the listing of TSTINDEX shows the creation of a simple address file, with 16 character search keys and (one line) addresses up to 8Ø characters long. The resulting records are then 96 bytes long.

```
PROGRAM tstindex;

TYPE
        key_t = ARRAY[1..256] of CHAR;
        rec_t = ARRAY[1..2048] of CHAR;
        ctrl_rec = RECORD
                c_1 : ARRAY[1..4] of INTEGER;
                rec_size : INTEGER;
                c_2 : INTEGER;
                key_size : INTEGER;
                end;
        index_record = RECORD
                disk : CHAR;
                filename : ARRAY[1..8] of CHAR;
                return_code : INTEGER;
                res_1 : INTEGER;
                ctl : ^ctrl_rec;
                reserved : ARRAY[1..196] of CHAR;
                END;

VAR
        key : key_t;
        rec : rec_t;
        cmd : CHAR;
        ir : index_record;
        tem_d : ARRAY[1..2048] of CHAR;

PROCEDURE INDEX0 ( command : CHAR;
                var key : key_t;
                var rec : rec_t;
                var ir : index_record ); extern;

PROCEDURE INDEX1 ( command : CHAR;
                var key : key_t;
                var rec : rec_t;
                var ir : index_record ); extern;
```

```
BEGIN   (* tstindex *)
ir := ' ';
write('Disk: ');
readln(ir.disk);
write('File: ');
readln(ir.filename);
REPEAT
        write('cmd: ');
        readln(cmd);
        cmd := upcase(cmd);
        key := ' ';
        rec := ' ';
        IF (cmd in ['A', 'D', 'R', 'U']) THEN
                BEGIN
                write('key: ');
                readln(key);
                IF (cmd in ['A', 'U']) THEN
                        BEGIN
                        write('data: ');
                        readln(tem_d);
                        rec := copy(key, 1, ir.ctl^.key_size) +
                               copy(tem_d, 1, ir.ctl^.rec_size -
                                        ir.ctl^.key_size);
                        END;
                END;
        (* justify or kompress must call INDEX1 *)
        IF (cmd in ['J', 'K']) THEN
                BEGIN
                rec := ' ';
                INDEX1(cmd, key, rec, ir);
                END
        ELSE
                INDEXØ(cmd, key, rec, ir);
        IF (ir.return_code <> Ø) THEN
                BEGIN
                writeln('Error:', ir.return_code);
                END;
        IF (cmd = 'Q') THEN
                writeln('query result: ', key[1]);
        IF (cmd in ['B', 'R', 'S']) THEN
                BEGIN
                writeln('key: ', copy(rec, 1, ir.ctl^.key_size));
                writeln('data: ', copy(rec, ir.ctl^.key_size + 1,
                        ir.ctl^.rec_size - ir.ctl^.key_size));
                END;
        UNTIL (cmd = '?');
END.
```

Execution of TSTINDEX is shown for a simple data base with 16 character names and up to 96 characters of information (which happen to be addresses). Note that the key length and record length are entered from the terminal in the N command.

```
A>EXEC B:TSTINDEX
Exec  ver 3.Ø

Disk: B
File: ADDRESS
cmd: W
cmd: N
Record size in bytes: 96
Key size in bytes: 16
Maximum number of records: 5ØØ
cmd: A
key: JRT
data: 'JRT Systems/45 Camino Alto/Mill Valley, CA 94941'
cmd: A
key: OLD
data: 'Old JRT Office/55Ø Irving St/SF, CA 94122'
cmd: B
key: JRT
data: JRT Systems/45 Camino Alto/Mill Valley, CA 94941
cmd: S
key: OLD
data: Old JRT Office/55Ø Irving St/SF, CA 94122
cmd: S
%INDEX error: Key not found
Error: 3
cmd: a
key: LITTLE
data: 'Little Italy/41Ø9 24th St/SF, CA 94114'
cmd: a
key: SZECHWAN
data: 'Szechwan Court/1668 Haight St/SF, CA 94117'
cmd: f
cmd: r
key: JRT
key: JRT
data: JRT Systems/45 Camino Alto/Mill Valley, CA 94941
cmd: r
key: OTHER
%INDEX error: Key not found
return code 3
cmd: z
```

```
cmd: ?
Error: 7

Program termination
```

7.5 CLOSE

Format
CLOSE (file_variable);

The CLOSE builtin procedure terminates processing against a disk The CLOSE builtin procedure terminates processing against a sequential or random disk file. If a sequential output file is not properly closed, the data written out will be lost because CLOSE updates the disk directory. This procedure also releases storage reserved for input/output buffers of sequential files.

Examples:

```
CLOSE ( F1 );
CLOSE ( DATA_FILE );
CLOSE ( MASTER_CUSTOMER_REPORT );
```

7.5 EOF

Format
EOF (file_variable);

The end of file function indicates when the end of a file is reached during input processing. It returns a Boolean value of true immediately after end of file detection, otherwise it returns false. The EOF function has no meaning in console or random disk processing.

When processing a file in text mode, end of file is detected when all data up to the first ctl-z (1AH) has been read. This is the standard character to indicate the end of data.

When processing a file in binary mode, end of file is detected when all the data in the last allocated sector of the file has been read.

Examples:

```
(* COMPUTE THE AVERAGE OF A FILE OF NUMBERS *)
RESET( F1, 'DAILY.SAL', TEXT, 4096);
TOTAL := 0;
COUNT := 0;
WHILE NOT EOF(F1) DO
        BEGIN
        READ(F1; DAILY_SALES);
        TOTAL := TOTAL + DAILY_SALES;
        COUNT := COUNT + 1;
        END;
AVERAGE := TOTAL / COUNT;
CLOSE( F1 );


(* WRITE A FILE TO THE PRINTER *)
SYSTEM( LIST );
RESET( F1, 'TEST.PAS', BINARY, 2048 );
READ(F1; CH);
(* INSTEAD OF USING EOF, WE DIRECTLY TEST FOR
A CHARACTER 1AH, SINCE THIS IS BINARY FILE  *)
WHILE CH <> CHR(1AH) DO
        BEGIN
        WRITE( CH );
        READ(F1; CH);
        END;
CLOSE( F1 );
```

7.6 EOLN

Format 1
EOLN (file_variable);

Format 2
EOLN;

The end of line function returns a Boolean value true if the end of line is reached otherwise false. This function applies only to console and text files, not to binary files.

Format 1 is used to sense end of line while reading disk files. Format 2 is used to sense end of line in console input.

This function is used primarily to read in an unknown number of data items from a line of text. Executing a READLN with or without any parameters, always resets EOLN to false and positions the file at the start of the next line of text.

Examples:

```
(* READ NUMBERS FROM CONSOLE, COMPUTE AVG *)
TOTAL := Ø;  COUNT := Ø;
WHILE NOT EOLN DO
        BEGIN
        READ( NUMBER );
        TOTAL := TOTAL + NUMBER;
        COUNT := COUNT + 1;
        END;
READLN;
AVERAGE := TOTAL DIV COUNT;

(* READ DATA FROM FILE, COUNT LINES OF TEXT *)
LINE_COUNT := Ø;
WHILE NOT EOF(F1) DO
        BEGIN
        READ(F1; DATA_ITEM );
        PROCESS_DATA( DATA_ITEM );
        IF EOLN(F1) THEN
                BEGIN
                LINE_COUNT := LINE_COUNT + 1;
                READLN(F1);
                END;
        END;
```

7.7 ERASE

Format
ERASE (filename);

The ERASE procedure deletes files from disk. It can be used to delete files from any available disk, by including the disk identifier in the filename.

ERASE is implemented as an external procedure. Any program referencing it must include its declaration:

```
PROCEDURE ERASE ( NAME : STRING[20] ); EXTERN;
```

Examples:

```
ERASE( 'TESTPGM.PAS' );

ERASE( CONCAT( 'B:', FILENAME, FILETYPE) );

ERASE( 'A:' + NAME + '.HEX' );

ERASE( BACKUP_FILE );
```

7.8 GET

Format

```
GET ( file_variable );
```

This standard Pascal procedure moves the next data item from the sequential file into the file's buffer variable. If there is not another data item in the file then the EOF function becomes true.

The READ procedure allows reading directly from a file into any variable.

```
READ ( F; X );
```

is equivalent to:

```
X := F^;
GET ( F );
```

7.9 OPEN

Format 1
OPEN (file_variable, filename, BINARY);

Format 2
OPEN (file_variable, filename, TEXT);

The OPEN builtin procedure is used to open files for random access. Format 1 is used to open files in binary mode. Format 2 is for text mode processing.

The file_variable refers to a file variable declared in the VAR declaration section. The filename is a string or structured expression which may include disk identifier letter.

The file specified by the filename is opened for use if present. If not present, a new file is created.

Both formats may be used with both RRN and RBA accessing.

Examples:

```
OPEN ( INVENTORY, 'INVENTRY.DAT', BINARY );

OPEN ( F1, RANGE + '.DAT', TEXT );

OPEN ( CASE_HISTORY, 'D:TORTS.LIB', BINARY );

OPEN ( DICTIONARY, 'B:SPELLING.LIB', BINARY );
```

7.1Ø PICTURE

The external function PICTURE allows you to format (real) numbers in powerful ways. Check printing is easy, as are commas within a number and exponential notation. Floating (or fixed) dollar signs are easy to specify. Credit and debit indications can be included. Literal characters such as currency signs can also be put in the formated string. COBOL and PL/I programmers will find familiar features such as with trailing signs.

PICTURE takes a format string and a real number as arguments. It returns a formated string, which can be printed on the console, the line printer, written to a file, concatenated with other strings, or saved for further processing. For example,

```
RES$ := PICTURE("*$##,###.##", 1456.2Ø);
WRITELN ("Sum: ", PICTURE("###,###.### ###",
        6583.1234567));
```

will set RES$ (which should be declared as a string or array of characters) to the eleven characters **$1,456.2Ø and write a line consisting of the twenty characters Sum: 6,583.123 456.

PICTURE is supplied as a compiled external function (the file PICTURE.INT). PICTURE must be declared in any program that uses it as

```
FUNCTION PICTURE (FMT : STRING; R : REAL) :
                        STRING; EXTERN;
```

The format string is not hard to create. PICTURE generally puts one character in the result string for every character in the format string, with the exceptions marked with a *. The format characters are summarized below.

Note that you will usually need only pound signs, commas, and periods in your formats.

Format	Replaced with
0	Literal zero (used only with exponential notation)
9	A decimal digit (always)
B	Space (or fill character)
CR	CR if the number is positive, else spaces
DB	DB if the number is negative, else spaces
E	Exponent (consisting of E, sign, and two digits) (*)
E+##	Exponent (sign and digit indications are ignored) (*)
L	Literal L (as a currency sign)
S	Minus or plus sign
V	Implied decimal point (*)
Z	Digit or fill character
-	Minus sign if number is negative, else space
+	Plus sign if number is positive, else minus sign
#	Digit or fill character
%	Digit or fill character
*	Asterisk fill
**	Asterisk fill and one digit
*$	Asterisk fill and floating dollar sign
**$	Asterisk fill, floating dollar sign, and one digit
,	Comma if digit has already been formated else space
/	Literal / (or fill character)
:	Literal : (or fill character)
space	Literal space (or fill character)
^	Exponent (E, sign, and two digits) (*)
^^^^	Exponent (*)
_	Next character is included literally (*)
	Next character is included literally (*)
_* or *	A single asterisk (*)
_$ or $	A single dollar sign (*)

Examples (our favorite formats)

-#.### ###^^^^	Large and small numbers
$##.##	Price of JRT Pascal
###,###	Number of happy customers
*$###,###.##	Checks (especially pay checks)
-##,###,###,###,###.##	Change in the national debt

In general, PICTURE can use any format with legal characters. It is possible to create ridiculous formats, such as "-+". An appropriate matching string will be returned (either space plus or minus minus in this case). If the format contains an invalid format character, PICTURE will complain and will return a two character string ??

Upper case and lower case letters are equivalent in the format, so E or e can be used for the exponent.

Simple number formating

Pound signs (#) are usually used to indicate where digits should be placed. A decimal point indicates where the decimal point should go. PICTURE does NO rounding, but just truncates insignificant digits. (The vertical bar just indicates the start of the result, and will not be included in the actual result.)

```
Format          Number     Result        Length
#####           15000      |15000          5
                -2.6       |   -2          5
                -17.98     |  -17          5
###.##          29.95      | 29.95         6
                -10.756    |-10.75         6
```

Punctuation

Commas can be inserted in the formated number. A comma in the format will cause a comma AT THE CORRESPONDING POSITION if a digit has already been put into the result. If no significant digit has been seen, then a space or asterisk is substituted. Note that PICTURE DOES NOT automatically put commas every third position. You can place commas in any meaningful (or meaningless) position in your number.

```
Format          Number     Result        Length
###,###         2470       |  2,470        7
#,###           -999       | -999          5
#,######        2743562    |2,743562       8
```

COUNT YOUR COMMAS and DIGITS. Commas can be used after the decimal point if desired.

A space (or B) works exactly the same as commas for those of you who want to punctuate numbers with spaces instead of commas. Note that this is different from the PRINT USING statement in Basics, which treat blanks as delimiters.

Exponential Notation

Exponential notation is indicated either with an uparrow (^) or the letter E. Following uparrows, signs, and digit indicators are ignored, so you can use ^^^^ or E+##. The formated exponent ALWAYS takes four characters: the letter E, the sign of the exponent, and two digits.

If you want PICTURE to create numbers in exponential notation with a leading Ø before the decimal point, you can use the digit Ø in a format before the decimal.

```
Format           Number     Result          Length

#.###^           15ØØØ     |1.5ØØE+Ø4          9
                 -2.5      |-.25ØE+Ø1          9
###.####^        15ØØØ     |15Ø.ØØØØE+Ø1      12
                 -2.5      |-25.ØØØØE-Ø1      12
###.####E+##     -2.5      |-25.ØØØØE-Ø1      12
Ø.### ###^^^^    15ØØØ     |Ø.15Ø ØØØE+Ø5     13
```

Signs

Normally, PICTURE will put a minus sign before the first significant digit in a number if that number is negative. This is called a floating sign, and will take up one digit position. You can have PICTURE handle the sign in many other ways. To put the minus sign (or blank) in a fixed position, use a - in the format. The minus sign can be before the first significant digit or at the end of the number.

To put a negative or a positive sign in a fixed position, use a plus sign (+) or an S instead of the minus sign.

```
Format           Number     Result          Length
-####            -12       |-  12             5
                 134       |  134             5
####+            -12       |  12-             5
                 134       | 134+             5
```

With exponential notation, you will generally want to specify the location of the sign, since a floating sign will cause one less digit before the decimal to be printed with negative numbers than with positive numbers.

```
Format              Number    Result           Length
-0.### ###^^^^      15000     | 0.150 000E+05  14
                    -15000    |-0.150 000E+05  14
-#.######^^^^       15000     | 1.500000E+04    13
-.###^              15001     | .150E+04         9
+.###^              15001     |+.150E+04         9
                    -2.506    |-.250E+01         9
.###-^              15001     |.150 E+04         9
                    -2.506    |.250-E+01         9
```

Note that you can put the sign in a number of inappropriate places and can even have the sign appear more than once.

Dollar signs and check printing

Floating dollar signs and asterisk fill work in a straightforward manner, and will produce the sort of results you would want for printing dollar amounts or checks. To enter a $ or * at a fixed position, use one of the "literal next" characters, the underline (_) or backslash () before the * or $.

```
Format              Number    Result       Length
_$##,###.##         2745.23   |$ 2,745.23    10
$##,###.##          2745.23   | $2,745.23    10
```

Note that the **, $$, and **$ formats are optional in JRT Pascal's PICTURE function. They are equivalent to *#, $#, and *$# respectively.

The only exceptions to the "one format character, one result character" rule are

1) the two "literal next" characters (_ and) which do not appear in the result
2) the V, which is not printed
3) the two exponent characters (^ and E) which always take four characters (and which cause following ^, +, -, #, and 9 specifi cations to be ignored in the format).

Overflow

Overflow occurs when the number to be formated cannot fit in the format provided, as when 1000 is to be formated in a three digit field (###). When that happens, PICTURE puts a % in place of all digits. In exponential notation, the only cause of overflow is with negative numbers when no sign is indicated and no digits are allowed before the decimal point.

Format	Number	Result	Length
-##	200005	| %%	3
######	-4000102	|-%%%%%	6
*$#,###	400102	|*$%,%%%	7
.###^	-207	|.%%%E+03	8

Testing formats for PICTURE

Here is a routine you can use to test your own picture specifications. (We use a extension of this program that allows file input and output to test ours.) The program reads the number of real numbers to be formated and the numbers to be formated. It then reads one format specification at a time and prints each of the numbers in that format.

```
PROGRAM TESTPICT;

CONST
     MAX_REAL = 100;

VAR
     I : INTEGER;
     NR_REALS : INTEGER;
     PIC : STRING;
     REAL_ARR : ARRAY[1..MAX_REAL] OF REAL;

EXTERN PICTURE (FMT : STRING; R : REAL) : STRING;

BEGIN
REPEAT
     WRITE('Number of real numbers to format: ');
     READLN(NR_REALS);
     UNTIL (NR_REALS < MAX_REAL);
FOR I := 1 TO NR_REALS DO
     READ(REAL_ARR[I]);
READLN;
```

```
     REPEAT
          WRITE('Format: ');
          READLN(PIC);
          IF (PIC <> '*') THEN
               FOR I := 1 TO NR_REALS DO
                    BEGIN
                    WRITELN(I:3, ' ',
                         REAL$(REAL_ARR[I]), '  |',
                         PICTURE(PIC, REAL_ARR[I]),
                         '|');
                    END;
          UNTIL (PIC = '*');
     END.
```

Note that currently, JRT Pascal requires that real numbers entered in exponential form must have a sign and two decimal digits. This restriction will be relaxed in the future.

Formats for ex-COBOL and PL/I programmers

The format character V can be used to set an implied decimal point without printing one. (V. and .V can also be used. The . will always be included in the result. Z can be used in place of #, and 9 can be used to force printing of a digit.

The "literal" / and : can be used. They will be replaced by the fill character (space or *) if appropriate. Multiple + and - signs can be used in place of # to cause floating signs.

Subtle differences between JRT Pascal's PICTURE and other languages will be found. Use the TESTPICT routine to experiment as needed.

7.11 PUT

Format

```
PUT ( file_variable );
```

This standard Pascal procedure appends the current value of the buffer variable to the sequential file.

The WRITE procedure allows writing directly to a file from any variable.

```
WRITE ( F; X );
```

is equivalent to:

```
F^ := X;
PUT ( F );
```

7.12 READ, READLN

Format 1 (console)

```
READ/LN ( variable1, variable2,... );
```

Format 2 (sequential disk)

```
READ/LN ( file_variable ; variable1, variable2,... );
```

Format 3 (random disk)

```
READ/LN ( file_variable, RRN, integer_or_real_expr ;
               variable1, variable2,...);
```

Format 4 (random disk)

```
READ/LN ( file_variable, RBA, integer_or_real_expr ;
               variable1, variable2,...);
```

The READ standard procedure is used to bring data from console or disk into main storage.

Format 1 is used for reading data from the console keyboard. When it is executed it will obtain data from the console buffer, convert to the proper format, and store the data in the specified variables. If sufficient data is not available, the system will wait for more data to be keyed in. If data is keyed in with an unacceptable format, a warning message is issued.

Dynamic string variables may only be used in READ format 1 - in console input, not in disk file input. To read character data from disk files, arrays of characters or records may be used.

Reading from the console into a dynamic string variable is treated differently. An entire line of text is obtained from the console and moved directly into the string variable. Separator characters and single quotes are ignored. The system will not allow more characters to be keyed in than can fit into the variable. The string variable must be the only variable in the READ's parameter list.

When all data on a given input line has been read in, the EOLN function becomes true. The READLN procedure has the additional purpose of reseting EOLN to false. READLN always clears out the current input line. For example, if 5 numbers were keyed in on one line and a READLN were issued with 3 variables in its parameter list, the last 2 numbers on that line would be lost.

Format 2 is used to read in data from a sequential disk file. Whether the file is processed as text or binary data is specified when the file is opened (RESET). The file_variable must refer to a file which has been successfully opened or a run-time error will occur.

Note that JRT Pascal uses a semicolon after the file_variable rather than a comma.

Format 3 is used to read in data from a random file by giving the relative record number (RRN) of the record required. The first record is at RRN=0. The file must have been successfully opened with the OPEN procedure. Sequential and random file accesses cannot be mixed unless the file is closed and re-opened in the other mode. The size of records on the file for RRN processing is determined when the file is declared. For example, a FILE OF REAL has a record size of 8 bytes.

Format 4 is used to read data from a random file by giving the relative byte address (RBA) of the data item required. The first byte of the file is at RBA=0. The file must have been successfully opened with the OPEN procedure. Random processing cannot be mixed with sequential processing but RRN and RBA processing can be mixed without re-opening the file.

Examples

```
READLN( A, B);

READ( DATA_FILE; X_DATA, Y_DATA );

READ( HISTORY_FILE, RRN, YEAR; MAJOR_EVENT );

READ( INQUIRY_FILE, RBA, 0; INDEX );

READLN;          (* RESET EOLN *)
```

7.13 RENAME

Format
RENAME (old_name, new_name);

The RENAME procedure is used to rename disk files on any disk. The old_name and new_name are string expressions.

RENAME is implemented as an external procedure. Any program referencing it must include its declaration:

```
PROCEDURE RENAME ( OLD, NEW1 : STRING[20] );
        EXTERN;
```

Examples:

```
RENAME( 'C:TEST.PAS', 'TEST2.PAS' );

RENAME( OLD_FILE_NAME, NEW_FILE_NAME );

RENAME( DISK + OLD_NAME, NEW_NAME );

RENAME( 'SORT.BAK', 'SORT.PAS' );
```

7.14 RESET

Format 1
```
RESET ( file_variable, filename, BINARY, bufr_size );
```

Format 2
```
RESET ( file_variable, filename, TEXT, bufr_size );
```

The RESET standard procedure is used to open already existing files for sequential input.

IMPORTANT CHANGE from version 2 to version 3 of JRT Pascal: RESET now set the EOF function to true and issues a warning message if the file does not exist on disk. It used to cause the program to terminate with an error. All programs should now test EOF immediately after RESET.

Format 1 is used to open files in binary mode. Format 2 opens files in text mode.

The file_variable refers to a file variable declared in the VAR declaration section. The filename is a string or structured expression which may include disk identifier letter.

The bufr_size is an integer expression which indicates the size of the input buffer to be allocated in dynamic storage. When storage is available, larger buffers are preferred because they result in fewer disk accesses and thus faster processing. The buffer size is rounded up to a multiple of 128.

Values like 1024, 2048, 4096 are recommended for bufr_size.

Examples:

```
RESET( INPUT_FILE, 'SOURCE.PAS', BINARY, 1024);

RESET( LOG, 'B:LOG.DAT', TEXT, 2048 );

RESET( DAILY_SALES, 'C:DAILY.DAT', TEXT, 256 );

RESET( STATISTICS, 'STAT.DAT', BINARY, 1024 );
```

7.15 REWRITE

Format 1
```
REWRITE( file_variable, filename, BINARY, bufr_size);
```

Format 2
```
REWRITE( file_variable, filename, TEXT, bufr_size);
```

The REWRITE standard procedure is used to open files for sequential disk output. A new file with the given filename is allocated. If a file with that name already exists, it is deleted to free the space allocated to it.

Format 1 is used to open files in binary mode. Format 2 opens files in text mode.

The file_variable refers to a file variable declared in the VAR declaration section. The filename is a string or structured expression which may include disk identifier letter.

The bufr_size is an integer expression which indicates the size of the input buffer to be allocated in dynamic storage. When storage is available, larger buffers are preferred because they result in fewer disk accesses and thus faster processing. The buffer size is rounded up to a multiple of 128.

Values like 1024, 2048, 4096 are recommended for bufr_size.

Examples:

```
REWRITE( LOG_FILE, 'F:LOG.DAT', TEXT, 512 );

REWRITE( REPORT, MONTH + '.RPT', TEXT, 1024 );

REWRITE( SYMBOL, PGM + '.SYM', BINARY, 256 );

REWRITE( STATISTICS, 'B:STATS.DAT', TEXT, 768);
```

7.16 WRITE, WRITELN

```
Format 1  (console)
WRITE/LN ( variable1, variable2,... );

Format 2  (sequential disk)
WRITE/LN ( file_variable ; variable1, variable2,...);

Format 3  (random disk)
WRITE/LN ( file_variable, RRN, integer_or_real_expr ;
                variable1, variable2,...);

Format 4  (random disk)
WRITE/LN ( file_variable, RBA, integer_or_real_expr ;
                variable1, variable2,...);
```

The WRITE standard procedure is used to transfer data from main storage to the console for display or to disk for storage.

Format 1 is used to write data to the console or printer. The console is always considered to be a text device, that is data is always converted to readable text format before output. Standard ASCII control characters are supported:

decimal	hex	purpose
9	Ø9h	horizontal tab
1Ø	Øah	line feed
12	Øch	form feed, clear screen
13	Ødh	carriage return, end line

For example, executing the Pascal statement WRITE(CHR(12)); will clear the screen of most types of CRT terminals.

The WRITELN statement is identical to the WRITE except that it also writes a carriage return character after the data, that is, it ends the current output line. A WRITELN may be used by itself, without any variables. This writes a blank line to the output device.

Format 2 is used to write data to sequential disk files. The file must have been successfully opened with a REWRITE procedure. This format may be used in either binary or text mode processing.

Note that JRT Pascal uses a semicolon after the file_variable rather than a comma.

Format 3 is used to write data to a random file by giving the relative record number (RRN) of the record being updated or created. The first record is at RRN=0. The file must have been successfully opened with the OPEN procedure. Sequential and random file processing cannot be mixed unless the file is closed and re-opened in the other mode. The size of records on the file for RRN processing is determined when the file is declared. For example, a FILE OF REAL has a record size of 8 bytes, the size of real variables.

Format 4 is used to write data to a random file by giving the relative byte address (RBA) at which the data is to be stored. The first byte of the file is at RBA=0. The data will be stored beginning at the specified RBA and continuing until it is all written out. The file must have been opened with the OPEN procedure. Random processing cannot be mixed with sequential processing but RRN and RBA processing can be mixed without re-opening the file.

When processing in text mode, a convenient formatting option is available. Any of the variables in the WRITE parameter list may be suffixed with a colon and an integer expression. This specifies the field width of the data value being written. If the data item is shorter than this then spaces will be inserted on the left of the item. This option is used when columns of figures must be aligned.

A second option is available for real numbers. After the field width integer expression, a second colon and integer expression may be used to indicate the number of digits right of the decimal place to be displayed.

Examples:

```
WRITELN( 'THE TIME IS ',GET_TIME );

WRITE( DATA_FILE; X[1], X[2], X[3] );

FOR I:=1 TO 100 DO
        WRITE( DATA_FILE; X[I] );

IF DATA < 0 THEN
        WRITE( NEGATIVE_DATA; DATA )
ELSE
```

```
        WRITE( POSITIVE_DATA; DATA );

WRITELN( REPORT; TOTAL_SALES:12:2 );

WRITE( CUSTOMER_FILE, RRN, CUST_NUM;
        NEW_CUSTOMER_RECORD );

WRITE( INQUIRY, RBA, Ø;  INDEX );

WRITELN;                 (* BLANK LINE *)

WRITE( CHR(ØCH) );       (* CLEAR SCREEN *)
```

8. Linker

The use of the linker is entirely optional. It is used to merge a Pascal program INT file with some or all of its external procedure/function INT files. It can process procedures written in assembler as well as Pascal. To run the linker enter:

```
EXEC LINKER
```

The linker will issue a prompt to the console for the program name. After the main program has been processed, you will be prompted to select which of the external procedures to merge. The procedures referenced by this program will be listed with their identification numbers (1 to 63). An asterisk indicates procedures selected. Possible replies to the 'Procedure selection' message are listed below. More than one number may be entered each time. Entering zero ends the interactive portion and causes merge processing to begin.

reply	purpose
1 to 63	select this procedure
-63 to -1	de-select this procedure
100	select all procedures
-100	reset, select none
0	end selection, begin processing

The output module file will have the same filename as the main program and a filetype of INT. The filetype of the main program input file will be renamed to IN2. If any of the selected input procedure files are not present a run-time error will occur and the linker will terminate. All files must be present on the A: disk.

9. Customiz

External procedures and functions are compiled separately from the main program. They can be linked together with the main program using the linker. If this is not done then they will be automatically loaded from disk into the computer's storage when they are first referenced. If a short-on-storage condition arises, they may be purged from storage if they are not currently active.

Procedures which are rarely used, like initialization or error handling, would not occupy main storage except when needed. Also very large programs might be divided into several phases, each corresponding to an external procedure.

The EXEC loads the external procedures from disk. There is no need to inform EXEC on which disk each procedure resides - it will search for them. This means that you do not have to put all the program sections on to the A: disk.

EXEC and the compiler JRTPAS3 contain 'disk search lists' which specifies which disks are available on the system. The default lists are set to 'AB'. The search lists should be modified to reflect your hardware configuration. The Customiz program is provided to modify the lists in both EXEC and JRTPAS3. To run Customiz enter:

```
EXEC CUSTOMIZ
```

You can enter the new disk search list with up to four disk letters specified. The letters must be contiguous. The list also determines the sequence in which the disks are searched for external procedures and functions.

10. Assembler

The JRT Pascal system provides two methods of preparing external procedures and functions written in assembly language. A special purpose assembler is provided which generates modules in the correct format. The second method may be used if a Microsoft format assembler is available such as RMAC or MACRO-80. The CONVERTM utility converts the REL files produced by these assemblers into INT format files which may be accessed as external procedures.

The JRT assembler translates 8080 assembly language into JRT relocatable format modules. These modules can be called from a Pascal program as if they were Pascal external procedures. Parameters may be passed to them and function return values may be received.

The JRT assembler is compatible with the standard ASM program distributed with CP/M. Input files have a file type of ASM. The assembler output is a file of type INT, which may be linked with the main program or automatically loaded at run-time.

10.1 Entry codes

After an external procedure is loaded into main storage, EXEC transfers control to it. A five byte code (95,6,0,92,0) is placed at the start of the procedure to inform EXEC that this is an assembler procedure rather than Pascal. The procedure must end with a return (RET) instruction. Any registers except the 8080 stack pointer may be modified.

Example of entry codes:

```
;procedure entry
        db 95,6,0,92,0  ;required entry codes
;
;send a message to console
        mvi c,9         ;print buffer code
        lxi d,msg       ;address of message
        call 5          ;bdos entry point
;
        ret             ;end of procedure
```

```
;
msg     db 'JRTASM sample procedure'
        db 0dh,0ah,'$'  ;carriage return
        end
```

If this procedure were named SAMPLE.ASM then the declaration in the Pascal program referencing it would be:

```
PROCEDURE SAMPLE; EXTERN;
```

10.2 Operating JRTASM

To assemble an external procedure enter:

```
EXEC JRTASM
```

You will be prompted at the console for the input filename and options. The options are:

1 - produce a listing on the console during pass 1 of the assembly process, useful for debugging

C - produce an output file of type 'COM' rather than 'INT', this is not an external procedure but a directly executable command file in standard CP/M format, an ORG 100H directive should be included since the default origin is 0

10.3 Directives

These assembler directives are supported:

directive	purpose
ORG	set location counter, not used in external procedures
SET	assign a value to a variable
EQU	assign a value to a fixed symbol
IF/ELSE/ENDIF	conditional assembly of code, may be nested to 16 levels
DB	define byte, multiple operands
DW	define word
DS	define storage

READ	used to assign a new value to a variable, like SET except that value is obtained from console
WRITE	display strings or expressions on console

Example of directives:

```
a       set 9
        if a = 9
        write 'a is equal to nine'
        else
        write 'a is not equal to nine'
        endif
;
x       read    ;msg at console will ask for x
        write 'x squared is ',(x * x)
;
a       set a + 1       ;increment a
        db 'string',a,255
;
```

10.4 Expressions

Integer expressions can be used in assembler instructions. Expressions are either fixed or relocatable. A symbol is relocatable if it refers to an address, otherwise it is fixed. If any symbol in an expression is relocatable then the entire expression is relocatable. Parentheses may be nested to any level.

These operators are supported:

```
* / + -
NOT AND OR XOR
MOD HIGH LOW
EQ NE LT LE GT GE
```

10.5 Parameters and function return values

Parameters of any data type may be passed to assembler external procedures and functions. The EXEC maintains a data stack which contains all static variables, parameters, function return values and procedure linkage blocks.

Three address pointers are used to access the data stack. These are available to external procedures in the 8080 register pairs on entry to the procedure.

```
BASE (HL) - address of the data stack
CUR  (DE) - address of the linkage block for
            currently active procedure
TOS  (BC) - top of stack, points past last
            allocated byte
```

```
          I               I
          I               I
   TOS-->I               I
          I---------------I
          I               I
          I    6 bytes    I  linkage block for
          I               I  current procedure
   CUR-->I               I
          I---------------I
          I    2 bytes    I  parameter length fld
          I---------------I
          I               I
          I    x bytes    I  parameters of
          I               I  current procedure
          I               I
          I---------------I
          I               I

          I               I
          I               I
          I               I  global variables
          I               I  of main program
          I---------------I
          I               I
          I    6 bytes    I  linkage block for
          I               I  main program
          I               I
  BASE-->I---------------I
```

With the three data stack pointers, the parameters passed to the procedure can be accessed. If it is a function the return value can be stored. Also the global variables of the main program can be accessed. For example, if the first global variable declared in the main Pascal program which calls the external procedure is an integer named INT1 then just add 6 to the BASE pointer to get the address of INT1. The BASE pointer is in register pair HL on entry to the procedure.

Data stack after procedure call DEMO('A',7);

```
'A'  7     length      linkage block
41  Ø7ØØ   Ø3ØØ       xx xx xx xx xx xx   yy
                      I                   I
                      CUR                 TOS
```

The two byte integer fields are in 8Ø8Ø byte-reverse format. The parameter length field is equal to three. The linkage block is six bytes of unspecified data.

Parameters are accessed by decrementing the CUR pointer. Pascal value parameters are actually present in the data stack. For reference parameters, the address of the variable is present in the data stack. If the procedure has no parameters, the parameter length field is zero.

Function return values must be stored just before the function's first parameter in the data stack.

Data stack after function call X := TEST(3,8); The return value is of type integer.

```
     3        8       length    linkage block
rrrr Ø3ØØ    Ø8ØØ     Ø4ØØ     xx xx xx xx xx xx   yy
I                              I                   I
return value                   CUR                 TOS
```

If the return value is of type CHAR, a string, or a structured variable (entire array, entire record) then there is a two byte length field between the return value and the first parameter. This field is set by EXEC and must not be modified. If the return value is a dynamic string, the current length field is a two byte field at the beginning of the string, this must be set to the desired length of the field.

Data stack after function call NAME:=LOOKUP('X',1); The return value is of type ARRAY [1..4] OF CHAR;

```
   return value rv len    'X'  1     length linkage block
   rr rr rr rr  0400      58   0100  0300 xx xx xx xx xx xx  yy
                                          I                  I
                                          CUR                TOS
```

10.6 Debugging assembler procedures

One effective way to debug external procedures written in assembler uses the CP/M Dynamic Debugging Tool DDT. If you are running a Pascal program under DDT then an RST 7 instruction will be seen as a breakpoint and allow you to use all of the DDT facilities. To run under DDT enter:

```
DDT EXEC.COM
Iprogram_name
G100
```

When the RST 7 instruction is encountered, DDT will gain control. The display, modify, disassemble facilities then can be used to examine the procedures data areas. To resume execution, use the XP command to set the instruction address ahead by 1, to get past the RST.

10.7 Convertm program

The convertm program translates Microsoft format REL files into JRT format INT files. Only REL files may be input - HEX files do not contain information about relocation addresses.

To run the convertm program enter:

```
EXEC CONVERTM
```

The program will inquire at the console for the name of the module to be translated. A file type of REL is assumed. The output module INT file is placed on the same disk.

10.8 Sample assembly programs

Three sample assembly programs are included here. Two external procedures (setbit, resetbit) and one external function (testbit) can be called from any Pascal program or external function. These small modules provide fast and simple bit manipulation facilities. They also illustrate the passing and returning of parameters for assembly language external procedures.

Listing of setbit.asm

```
;setbit.asm
;external procedure which sets a bit on in a byte
;
; procedure setbit ( var x : char; bit : integer );
;               extern;
; bit# in range Ø..7
;
;entry code
        db 95,6,Ø           ;int vmcode
        db 92               ;lpn vmcode
        db Ø                ;mode vmcode
;on entry  bc=wtos  de=wb  hl=wbase
;
;get bit# in b_reg,  addr(x) in hl,  x into c_reg
setbit  xchg                ;hl=wb
        dcx h! dcx h! dcx h! dcx h
        mov b,m             ;bit#
        dcx h! mov d,m! dcx h! mov e,m ;addr(x)
        xchg                ;hl=addr(x)
        mov c,m             ;c=x
;create mask
        inr b               ;incr loop count
        mvi a,1
loop    rrc
        dcr b
        jnz loop
;a=mask  c=byte
        ora c
        mov m,a             ;store byte
        ret
;
        end
```

Listing of resetbit.asm

```
;resetbit.asm
;external procedure which reset bit in a byte
;
; procedure resetbit ( var x : char; bit : integer );
;               extern;
; bit# in range Ø..7
;
;entry code
        db 95,6,Ø       ;int vmcode
        db 92           ;lpn vmcode
        db Ø            ;mode vmcode
;on entry  bc=wtos  de=wb  hl=wbase
;
;get bit# in b_reg,  addr(x) in hl,  x into c_reg
resetbit xchg           ;hl=wb
        dcx h! dcx h! dcx h! dcx h
        mov b,m         ;bit#
        dcx h! mov d,m! dcx h! mov e,m ;addr(x)
        xchg            ;hl=addr(x)
        mov c,m         ;c=x
;create mask
        inr b           ;incr loop count
        mvi a,Øfeh
loop    rrc
        dcr b
        jnz loop
;a=mask  c=byte
        ana c
        mov m,a         ;store byte
        ret
;
        end
```

Listing of testbit.asm

```
;testbit.asm
;external function which returns bit value of a byte
;
; function testbit ( x : char; bit : integer ):
;                boolean; extern;
;
; bit number is in range 0..7
;
;entry code
        db 95,6,0       ;int vmcode
        db 92           ;lpn vmcode
        db 0            ;mode vmcode
;on entry  bc=wtos  de=wb  hl=wbase
;
;get bit# into b_reg and x into a_reg
testbit xchg            ;hl=wb
        dcx h! dcx h! dcx h! dcx h ;point to bit lownib
        mov b,m         ;low byte of bit
        dcx h! mov a,m  ;x
        inr b
;shift loop
loop    rlc
        dcr b
        jnz loop
        jc true         ;bit is set
;false : bit is zero
        dcx h! mvi m,0! dcx h! mvi m,0
        ret
;true : bit is one
true    dcx h! mvi m,0! dcx h! mvi m,1
        ret
;
        end
```

11. Storage management

This section discusses the initialization and structure of main storage in the JRT Pascal system during execution of Pascal programs.

11.1 Main storage

When a Pascal program is started by entering the command "EXEC prog_name" the EXEC.COM file is loaded into main storage at address 100H by the CP/M operating system. After EXEC receives control from CP/M it determines how much storage is available and formats this area. EXEC then loads the Pascal program module from disk. Processing of the Pascal program then begins.

During program execution there are four main regions of main storage. Starting from the lowest address these are:

1. EXEC - the run-time environment, this region is fixed in size and contains the primary run-time support system

2. Pascal program module - fixed in size, this is the compiled Pascal program from an INT file

3. Data stack - variable in size, this region begins at the end of the Pascal program and grows toward higher addresses; this region contains all static variables (those created by VAR declarations), parameters passed to procedures and procedure activation blocks

4. Dynamic storage - variable in size, this region begins at the top of available storage and grows down toward lower addresses; this region contains dynamic variables (those created by the NEW procedure), input/output buffers, file control blocks, external procedures and EXEC control tables

Since the data stack and dynamic storage regions grow toward each other, a collision between these areas is possible when storage is nearly full. To prevent this condition the run-time system maintains a 64 byte cushion between the two areas. When the distance between them becomes less than 64 bytes the run-time system takes several actions to restore the cushion. If there is less than 64 bytes of free space in main storage, the least-recently-used procedure will be deleted. Dynamic storage is then compressed (see section 11.2). Processing will continue even if the cushion cannot be restored, although performance will gradually decrease. Only if there is actually a collision between the data stack and dynamic storage will the run-time system recognize an error condition and terminate processing.

Map of main storage use in the JRT Pascal system.

```
high     -------------------------
address I  dynamic storage       I
        I                        I
        I  variable in size      I
        I       direction    l   I
        I       of growth    l   I
        I                    V   I
        I------------------------I
        I  unused area           I
        I------------------------I
        I  data stack            I
        I                        I
        I  variable in size      I
        I       direction    A   I
        I       of growth    l   I
        I                    l   I
        I------------------------I
        I  Pascal program        I
        I  INT module            I
        I                        I
        I  fixed in size         I
        I------------------------I
        I  EXEC                  I
        I  run-time system       I
        I                        I
        I  fixed in size         I
low     I                        I
address -------------------------
100H
```

11.2 Dynamic storage

The JRT Pascal run-time system provides true dynamic storage with auto-compression and for external procedures, virtual storage is supported.

The JRT Pascal Dynamic Storage Management System is designed to provide complete support for advanced features such as dynamic data structures (linked lists, trees, rings,...) and completely automatic virtual storage for external procedure and function code. Dynamic storage may contain these items:

1. external procedures/functions
2. dynamic variables created by the NEW procedure
3. input/output buffers
4. file control blocks
5. EXEC control blocks and pointer tables
6. a free list of deallocated storage blocks

All of these items are allocated as blocks of dynamic storage. Dynamic storage blocks are addressed indirectly in JRT Pascal in order to allow the blocks to be moved during compression by updating a pointer table. The value stored in a pointer variable by the execution of the NEW procedure is a "virtual address" rather than the real address of the block allocated. The virtual address is used to locate an entry in an internal table called a pointer table, which contains the size and real address of each storage block. There may be up to 32 pointer tables and each one contains up to 52 entries for storage blocks. During dynamic storage compression, the real address of a storage block may change but the virtual address does not change.

The dynamic storage manager performs these services.

1. format dynamic storage and initialize pointer tables

2. maintain the free list - this is a linked list which contains blocks of storage which have been deallocated by the DISPOSE procedure, by closing a file or by purging of an external procedure

3. allocate a storage block - when a storage block is requested by the NEW procedure, opening a file or loading an external procedure, the storage manager attempts to satisfy

this request by searching the free list or extending the dynamic storage region; when scanning the free list for a block, the first block which is large enough is selected; if this block is much too large, it is split and the remainder returned to the free list; after a block has been found, its real address, size and a flag field are entered in a pointer table

4. release a block of storage - add a deallocated block to the free list and delete the corresponding pointer table entries

5. determine the amount of free space - the free space is the sum of the sizes of all blocks on the free list and the size of the gap between the data stack region and the dynamic storage region

6. compress dynamic storage - All of the allocated storage blocks are moved into the top of storage to eliminate free space. The free list is set to a null pointer. The pointer table entries of all blocks are updated. If external procedures were moved then their relocatable addresses are adjusted. If active external procedures were moved then the Pascal program counter and the procedure return addresses are adjusted.

7. convert the virtual address of a block to a real address

12. External Procedures and Functions

External procedures are a facility for segmenting programs into separately compiled modules. With these, the size of the entire program can be practically unlimited. This is because, unlike with segment procedures, overlays or chaining, the virtual storage manager loads and when necessary deletes program sections all automatically. This makes the actual storage of the computer seem much larger than it really is.

Refer to the section on storage management for a full description of virtual/dynamic storage.

External procedures are loaded into dynamic storage by EXEC when they are first referenced, unless they were linked with the main program to form one module. The loading is transparent to the programmer in that no planning or effort is required.

External procedures remain in storage unless a short-on-storage condition occurs, then the least-recently-used procedure may be deleted. If this happens, the control blocks associated with the procedure are kept so that re-loading, if necessary, could be done more rapidly. When main storage is severely overloaded, frequent deleting and reloading of external procedures may occur. This condition is called "thrashing." Thrashing can be recognized by unusually frequent disk accessing and little useful processing being done by the program. It is necessary in this case to reduce the storage requirements of the program.

12.1 Coding external procedures and functions

The external procedure Pascal file is very similar to a standard "internal" procedure in format. In many cases the only differences from a standard procedure format are that the PROCEDURE reserved word is preceded by the reserved word EXTERN and that the whole file is ended with a period to signify the end of the compile unit. An example of this basic case follows.

```
EXTERN

(* PRINT THE TOTAL AND AVERAGE OF 4 NUMBERS *)
PROCEDURE XDEMO ( A,B,C,D : REAL );
VAR
TOTAL : REAL;

BEGIN
TOTAL := A + B + C + D;
WRITELN('TOTAL =',TOTAL,
        '  AVERAGE =',TOTAL / 4.0);
END; .
```

JRT Pascal external procedures can access all of the global variables in the main program. The global variables are those in the main program declared before any procedure or function declarations. They are variables that are available globally not only local to some procedure. In the preceding example, TOTAL is a local variable - it is not accessible outside of the procedure XDEMO.

To access global variables or files, their declarations are inserted in the external procedure file after the reserved word EXTERN and before the procedure header. The three declaration sections CONST, TYPE, VAR may be inserted at this point. They must be identical to the global declarations in the main program, except that additional constants and type identifiers may be added here.

Type identifiers may be required in the procedure header parameter list or in a function return value declaration. The declaration of these type identifiers should appear in the same location as the global declarations - just after EXTERN.

```
EXTERN

CONST

NAME_SIZE = 32;

TYPE

NAME = ARRAY [1..NAME_SIZE] OF CHAR;

CUSTOMER_RECORD = RECORD
        CUST_NAME, CUST_ADDR    : NAME;
        BALANCE                 : REAL;
        END;

VAR     (* MAIN PROGRAM GLOBAL VARIABLE *)

CUSTOMER_LIST : ARRAY [1..100] OF
                        CUSTOMER_RECORD;


(**** SEARCH CUSTOMER LIST FOR GIVEN NAME ****)
FUNCTION SEARCH ( N : NAME ) : CUSTOMER_RECORD;
VAR
I : INTEGER;

BEGIN
I:=1;

WHILE  (N <> CUSTOMER_LIST[I].CUST_NAME)
       AND (I <= 100) DO   I:=I+1;

IF  N = CUSTOMER_LIST[I].CUST_NAME THEN
        SEARCH:=CUSTOMER_LIST[I]
ELSE    SEARCH:=' ';

END; .
```

12.2 Referencing external procedures and funtions

External procedures and functions must be declared in the main programs which reference them. Their declaration is identical to a regular procedure except that the entire body of the procedure is replaced with the reserved word EXTERN.

```
PROCEDURE PLOTTER ( X,Y : INTEGER ); EXTERN;

FUNCTION CUBEROOT ( A : REAL ): REAL; EXTERN;
```

For clarity it is useful to group all external procedure declarations as the first procedure declarations in the program. External procedures may reference other external procedures, if appropriate declarations are included in the referencing procedure.

EXEC identifies external procedures by a sequence number. External procedures should always be declared in the same sequence - in main program or in another external procedure.

Note that the user must ensure that external procedure declarations and parameter lists are consistent among different files, since the compiler does not validate this.

13. Debugging Pascal programs

Debugging computer programs is the process of correcting "bugs" in a program so that it will perform as desired. There are two phases of debugging - correcting syntax errors in a program in order to obtain an error free compile and correcting errors which occur during the running of the program after a clean compile. Referencing an undeclared variable is an example of the first kind of error. Dividing by zero is an example of the second kind. This section is primarily concerned with the second kind of error - those that occur during program testing.

JRT Pascal provides several facilities to simplify the location and the correction of run-time errors. The debugging philosophy is to provide the programmer with as much relevant information as possible in a clearly formatted display. The run-time system detects errors at two levels of severity - errors and warnings. When warnings occur, a message is issued and processing continues. When an error occurs processing must terminate.

Error and warning messages are all presented in verbal format - there are no number or letter codes to look up. These messages are stored on a disk file so main storage is not wasted.

13.1 Trace options

JRT Pascal allows a trace of the program line numbers while a program is running. This trace may be turned on or off by the program itself. The range of line numbers to be traced may also be set by the program.

A trace of procedure names can also be produced. On entry to each procedure, the name and activation count is displayed. On exit, the name of the procedure is displayed. This feature can also be turned on or off under program control.

The Exec interrupt mode can be entered by entering a control-n while a program is running. In this mode the traces and line number range can be modified. Other system status information can also be displayed. When in interrupt mode, entering a space character will cause a list of valid commands to be displayed.

Exec interrupt allows asynchronous control of the trace facility. Programmed control is also supported with the SYSTEM builtin procedure.

An interactive external procedure to control these trace facilities at run-time is provided. The DEBUG procedure is described in section 13.2.

To use these traces, the %LTRACE and %PTRACE compiler directives must be inserted in the program. It is recommended that the first line of a program being tested contain both directives, so that the entire program will be subject to tracing. An additional advantage is that when these options are present, if an error or warning occurs, the line number and latest procedure name will be displayed with the error message.

The coding of these directives and use of the SYSTEM builtin procedure to control the traces are described in the section on compiler directives.

13.2 DEBUG procedure

The DEBUG external procedure allows the control of the dynamic trace facilities while a program is being tested. The procedure and line traces can be turned on or off and the line range can be set by commands entered from the console.

The file DEBUG.INT on the distribution disk, is the compiled external procedure module. To reference an external procedure from a Pascal program, it is necessary to declare it:

```
PROCEDURE DEBUG; EXTERN;
```

The procedure can be called from any number of places in the test program by inserting a procedure call statement:

```
DEBUG;
```

When it is activated, DEBUG will interact with the programmer to modify the current trace operations.

Listing of DEBUG.PAS

```
extern

procedure debug;

var
reply : char;
lower, upper : integer;

begin (* debug *)
writeln;
write('Activate line trace?  y/n : ');
readln(reply);
if upcase(reply) = 'Y' then
        begin
        write('Range of lines?  lower,upper : ');
        readln(lower,upper);
        system( ltrace );
        system( lrange,lower,upper );
        end
else    system( noltrace );

write('Activate procedure trace?  y/n : ');
readln(reply);
if upcase(reply) = 'Y' then system( ptrace )
else system( noptrace );
writeln;
end; (* debug *).
```

13.3 System status display

When an error is detected, an error message is displayed on the console. The current line number and last entered procedure name may also be displayed (see section 13.1). A system status display is also created - this contains useful information about the current state of the run-time system.

The system status display shows nine fields of information. If external procedures are present, the external procedure table is also formatted and displayed.

System status display

```
addr :54F5   prog :3BA7   size :4815
base :83BC   cur  :89AC   tos  :8A33
low  :A8B9   compr:0002   purge:0000
```

Most of these values indicate the use of storage in the run-time system. Storage management is discussed fully in another section - a simplified map of storage is presented here.

```
             ------------------
             I    CP/M        I
             I----------------I
             I  dynamic       I
             I  storage       I
   low--->   I----------------I
             I                I
             I  unused        I
             I                I
   tos--->   I----------------I
             I                I
   cur--->   I  data stack    I
             I                I
   base-->   I----------------I
             I                I
             I  Pascal code   I <--addr (of error)
             I                I
   prog-->   I----------------I
             I                I
             I  EXEC run-time I
             I  system        I
   100h-->   I----------------I
             I  reserved area I
             ------------------
```

1. addr - the address at which the error occured, may be in Pascal code or in dynamic storage area if error was in external procedure

2. prog - the starting address of the main Pascal program

3. size - the size of the main program module

4. base - the base or bottom of the data stack

5. cur - the address of the current procedure activation block

6. tos - top of stack, the address just past the end of the data stack

7. low - the lowest address occupied by any dynamic storage block

8. compr - a count of the number of times storage has been auto-compressed

9. purge - a count of the number of external procedures that have been purged from dynamic storage due to short-on-storage condition

The system status display may contain one additional line of input/output information. The name of the most recently referenced file, a status byte and the current default disk will be displayed if files have been used by the program.

```
@:SAMPLE  PAS   88   A
```

If the file was opened without specifying a disk letter then @ is shown otherwise the disk letter. The status byte contains several flag bits:

bit	meaning
8Ø	file is open
4Ø	random mode - not sequential
2Ø	text mode - not binary
1Ø	EOLN flag set
Ø8	input - not output or random
Ø4	EOF flag set

Formatted external procedure table

exproc name	addr	use cnt	time	stat
ACCTPAY1	C2AE	ØØØØ	ØØØ4	3Ø
ACCTPAY2	3E22	ØØØØ	Ø165	74
GENLEDG1	ØØØ1	ØØØØ	ØØØØ	ØØ
ACCTREC1	3F55	ØØØ1	Ø14E	F4
ACCTREC2	44ØC	ØØØ1	Ø15A	F4
SORT	ØØØ1	ØØØØ	ØØØØ	ØØ
+INVENTRY	5Ø3A	ØØØ1	Ø2ØD	F4
CHECKS	5Ø52	ØØØØ	Ø1Ø3	3Ø

1. exproc name - the name of the external procedure or function, a plus sign indicates the external procedure which was most recently entered or exited, this is not necessarily the currently active procedure

2. addr - the address in main storage of the external procedure module, if this value is ØØØ1 then the module is not currently in main storage

3. use cnt - a count of the number of times the procedure is CURRENTLY active, usually this will be ØØØØ (not active) or ØØØ1 (active), it will be greater than ØØØ1 only if the procedure is called recursively

4. time - in order to determine which procedure was least-recently-used, the run-time system maintains a pseudo-timer which is incremented once on each entry to or exit from an external procedure - the time field contains the value of the pseudo-timer the last time the procedure was entered or exited

5. stat - a status indicator with several flag bits:

bit	meaning
8Ø	procedure is currently active
4Ø	procedure was linked with main program
2Ø	procedure is currently in storage
1Ø	procedure file control block is open
Ø4	procedure address is real, not virtual

13.4 Run-time messages

The run-time system provides several messages to aid in the correction of error or exceptional conditions. In addition to these general messages, about 75 more specific messages of 1 to 4 lines of text are provided to describe particular error conditions.

The general run-time messages are all prefixed with a % character. These messages are listed here:

%Entry - indicated entry to a procedure when procedure trace is active, procedure name and activation count are listed, external procedures are indicated by an asterisk before the name

%Error - fatal error detected by run-time system, program terminates

%Exit - indicates exit from procedure when procedure trace is active, procedure name is listed, external procedures are indicated by an asterisk before the name

%Extern - indicates that error occured while attempting to load an external procedure module, the procedure name is listed

%Input error - indicates a format error when reading console input, such as entering a character string when an integer was expected

%Line - indicates line number where error occured, module must have been compiled with %LTRACE option

%Main - error occured in main program BEGIN-END block, not in procedure

%Proc - error occured in procedure, not in main program BEGIN-END block

%Trace - line number trace indicator

%Warning - non-fatal error condition, processing continues

13.5 Common Problems

A. General difficulties

1. The master disks accidently got erased by a program... MAKE BACKUP COPIES OF JRT PASCAL when you first get it. May we suggest: Remember the Master Disk, to keep it wholly. As a read only disk. Please.

2. The disks will not boot up when one is put in drive A and the system is reset... After you copy JRT Pascal to your own working disks, put a copy of YOUR operating system (using SYSGEN or whatever YOUR operating system calls it) on the working disks. We cannot put your operating system on disks we distribute.

3. With CP/M 1.4, C/DOS or the equivalents, CUSTOMIZ, LINKER, and random i-o in general will not work... Sorry about that, but to get random i-o on 8 megabyte files, CP/M 2.2 would be required. LINKER is never required for JRT Pascal. The function of CUSTOMIZ can be performed by two simple patches in DDT. This involves patching the disk search list in EXEC.COM and JRTPAS3.COM. Both lists are at 155 hex and consist of up to four upper case letters followed by a Z.

```
A>DDT EXEC.COM
-S155
Ø155 41 41
Ø156 42 42
Ø157 4A 5A       (an upper case Z)
Ø158 ØØ .
-GØ
A>SAVE ____ EXEC.COM
```

For JRTPAS3.COM, the SAVE command is

```
A>SAVE ____ JRTPAS3.COM
```

4. The diagnostic "JRTPAS3?" or "SOURCE FILE NOT FOUND" comes up... CP/M needs to know the drive on which the file to be run is located, if it is not the current default drive. JRTPAS3 needs to know the drive on which the source file to be compiled is located. Further, that source file must have a .PAS suffix on the name. So, for example, you may need to type B:JRTPAS3 B:PGM if the default drive is A: and both JRTPAS3 and PGM.PAS are on

the B: drive.

5. The compiler and everything else does not fit on one disk... There are many possible ways to set JRT Pascal up when you have a system with small drive capacities. One is:

On disk A:	On Disk B:
- EXEC.COM	- JRTPAS3.COM
- your editor (ED, Wordstar, etc.)	- PASCALØ.INT
- the PASCAL source program being developed	- PASCAL1.INT
- perhaps PASCAL.LIB	- PASCAL2.INT
	- PASCAL3.INT
	- PASCAL4.INT
	- PASCAL.LIB

You Osborne owners may need to do some shuffling until you find the the arrangement that works best for you. For example, the compiler disk could be on drive A:, which would alternate with the Wordstar disk as necessary (with appropriate control-C's after disk changes). The source and object programs could then stay on B:, perhaps with EXEC.COM and another copy of PASCAL.LIB.

Be sure there is a copy of your operating system on each disk you put in drive A.

6. The compiler (or run-time) used to work, but now it doesn't... Use EXEC VERIFY to check the compiler and/or run-time files again. Even if the sums agree, a file or files may have gotten shuffled by a malfunctioning program, hardware errors, or bad diskette handling. If necessary, go back to the original master disks and copy the needed files to a new diskette. If necessary, act as if you just got JRT Pascal.

7. EXEC VERIFY does not even work... Make sure that EXEC.COM, VERIFY.INT, and PASCAL.LIB are MOUNTED on your disk system, and that you told CP/M the right drive for EXEC.COM and that you gave EXEC the right location for VERIFY.INT. You may need to use B:EXEC B:VERIFY if the files are on B:. Remember when you run EXEC.COM that PASCAL.LIB must be present.

8. BDOS errors show up when a DIR is requested of a master disk... Make sure that you system is expecting a disk in the format provided. For example, single density 8". Some operating systems can not sense a density change or disk format change once they have determined "the format for

that drive". A system reset may be needed.

B. Compiler Errors

1. String literal too long... Somewhere in the program, a literal string does not have a closing (or opening single quote). This error is caught by the lexical scanner before the program is listed. (Most editors make it easy to search for all lines with single quotes).

2. Block structure invalid (and other strange diagnostics)... Perhaps the program is attempting to declare or use a reserved word. The list of reserved words in JRT Pascal is somewhat longer than standard. For example, LENGTH and POS are reserved.

3. Compiler acts like something is not there... Many versions of Wordstar will set the high-order bit of the 'current' character when a file is closed, even when editing in non- document mode. ALWAYS end an edit with (^QC) before (^KD). Also, use PIP newfile.PAS=oldfile.PAS[Z] to clear off parity bits.

4. Compiler "goes away"... Hit system reset, then look for undeclared variables, types, or constants in the next line not listed. Also check for ; or , used inappropriately.

5. Out of memory... Split the program into a main program and external procedures so that each portion is 600 to 1200 lines long. (Maximum length depends on the program and the available memory.)

6. Array out of bounds at end of compilation... External procedure names can be 8 characters long and should not contain $ or _ characters, since the exproc name is turned into a CP/M file name.

C. Run-time Errors

1. Object file not found... Make sure that the source program is compiled successfully, and that the appropriate drive is indicated on the file name, as EXEC B:PGM.

2. Library not present... PASCAL.LIB must be present on one of the drives in the "disk search list" (usually A: or

B:).

3. Files never get written to... CLOSE(file_variable) is required after files have been written, so that CP/M performs a proper close on the file. Otherwise, the file size will be the next lower multiple of 16K in size. Usually zero.

4. Reading characters from a file, most of the characters in a word get skipped... The difference between binary and text modes are significant. If you want every character in a file, use binary mode in the reset or open statement.

5. Reading from a file in binary mode, end of file is hard to determine... Control-Z (1ah) marks the end of a text file (unless the real end of file on a 128 byte boundary occurs). Text for both character = CHR(26) and EOF. For binary records, a special record of all 255 (0ffh) or all EOF's (1ah) may be needed to mark the end of the file, since CP/M only keeps track of 128 byte sectors.

6. External procedures get all mixed up... Declare external procedures properly. When external procedures refer to other external procedures, the declaration order count must match those in the main program.

If your main has

```
FUNCTION COS(R : REAL): REAL; EXTERN;
FUNCTION SIN(R : REAL): REAL; EXTERN;
```

and your exproc has declared only

```
FUNCTION SIN(R : REAL): REAL; EXTERN;
```

lo and behold, the exproc will get a value of 1.0 if it passes 0.0 to what it thinks is SIN. The exproc will have actually called COS. Internally, external procedures refer to other external procedures by number. `Dummy` declarations such as PROCEDURE X1; EXTERN; can be used as place holders, as long as the names are unique. The name used in the MAIN program will be used to find the external procedure on the disk.

7. Values are not returned correctly from external functions (or arguments are not passed correctly to external procedures)... Make sure the declaration of arguments in

the calling program match those in the external procedure. If a VAR is missing in one and present in another, you could have trouble.

8. Control-C does not stop a program (or control-N does not stop it either)... Use control-N to cause an execution interrupt (you can either exit the program with Z or continue with R as appropriate). Use %LTRACE or $L when compiling the program to allow execution interrupts and also error diagnostics with line numbers.

14. Extended CASE statement

Format

```
CASE selector_expression OF
label_expression ... ,label_expression : statement;
 ...
 ...
ELSE : statement;
END
```

The CASE statement is used to select one of several statements for execution based on the value of the selector_expression. The selector_expression and the label_expressions must be of compatible data types.

The label_expressions are evaluated sequentially. If one is found equal to the selector, the corresponding statement is executed. If none are equal then the optional ELSE clause statement is executed.

The ELSE clause is a JRT Pascal extension. Also, standard Pascal allows only constants as labels, while expressions are allowed here. Not more than 128 label clauses are allowed in one CASE statement. Not more than 128 labels per label clause are allowed. The statements should be followed by a semicolon. The semicolon is optional on the last statement in the CASE statement.

Examples:

```
CASE I OF
2 : WRITELN('I IS 2');
4 : WRITELN('I IS 4');
ELSE : WRITELN('I IS NOT 2 OR 4');
END;
```

```
CASE LANGUAGE OF          (* STRING EXPRESSION *)
'PASCAL'   : YEAR := 1970;
'PL/I'     : YEAR := 1964;
'BASIC'    : YEAR := 1965;
END;
```

```
(* EXAMPLE OF EXPRESSIONS IN LABELS *)
CASE ANGLE OF
PHI             : WRITELN('PHI');
2.Ø * PHI       : WRITELN('TWO PHI');
3.Ø * PHI       : WRITELN('THREE PHI');
ELSE            : WRITELN('ANGLE NOT ON NODE');
END;
```

```
(* EXAMPLE OF BOOLEAN SELECTOR AND LABEL EXPRESSIONS *)

(* CHECK VOLTAGE V FOR VALID RANGE *)
CASE TRUE OF
(V > 2.5) AND (V < 4.3)          : PROCESS_RANGE_1;
(V > 5.6) AND (V <= 14.Ø8)       : PROCESS_RANGE_2;
(V > 35.6) AND (V <= 1ØØ.Ø)      : PROCESS_RANGE_3;
ELSE : WRITELN('VOLTAGE OUT OF VALID RANGES:',V);
END;
```

15. CRT Formatting

This section describes JRT Pascal CRT formatting facitlites. It requires a basic knowledge of Pascal and of JRT Pascal external procedures.

The CRTMAP utility enables the user to quickly format a CRT terminal screen. One record at a time may be displayed.

The utility program takes as its input a Map Description File (MDF) which describes the CRT map in a simple command language. The utility generates the source program for a Pascal external procedure which may then be compiled. This external procedure contains all the logic to display all or part of one record data type. Descriptive information may also be displayed on the screen.

Source code for CRTMAP is included and its features may be modified or extended. The distributed version of CRTMAP assumes a Televideo display terminal. It may be adapted to any other terminal or computer by modifying two lines in the program. These lines specify the control codes for cursor positioning and clearing the screen. Consult your Display terminal user manual for the codes for your system. The cursor positioning code is in procedure GOTOXY in the CRTMAP.PAS file. The screen clear code is procedure CLEAR.

Procedure PART2 from CRTMAP.PAS is reproduced here. This code generates "part2" of the generated external procedure. The line marked XXX contains the terminal codes for clearing the CRT screen. The line marked YYY contains the terminal codes for moving the cursor to a particular position.

```
procedure part2;
begin
writeln(f2; 'procedure clear;');
writeln(f2; 'begin');
writeln(f2; 'write(chr(27),''*'');');              { XXX }
writeln(f2; 'end;');
writeln(f2);
writeln(f2; 'procedure gotoxy ( x,y : integer );');
writeln(f2; 'begin');
writeln(f2; 'write(chr(27),''='',chr(y+2Øh),chr(x+2Øh));'); { YYY }
writeln(f2; 'end;');
writeln(f2);
end; {part2}
```

The CRT screen coordinates have the origin Ø,Ø in the upper left corner.

```
     Ø        X            79
     -------------------------
   Ø I                        I
     I                        I
     I                        I
   Y I                        I
     I                        I
     I                        I
  23 I                        I
     -------------------------
```

The first coordinate X indicates the column, the second Y indicates the row.

15.1 Structure of the external procedure

CRTMAP generates a Pascal external procedure according to the parameters in the Map Description File. This external procedure then does the display formatting of your data record.

Structure of the generated external procedure

```
PART1    EXTERN
         TYPE
         %INCLUDE type_declaration_filename
         PROCEDURE exproc_name
                 ( VAR R : type_name );

PART2    PROCEDURE CLEAR;
         PROCEDURE GOTOXY;

PART3    PROCEDURE DISPLAY;      { format the CRT }

PART4..PART8 (omitted)

PART9    BEGIN
         main_line_code
         END;.
```

15.2 Map Definition File

The MDF defines the format of the CRT screen for the display of one record type. CRTMAP recognizes seven different MDF commands.

The MDF commands MUST be entered in a fixed sequence except for LITERAL and FIELD which may be intermixed. There should be one command per line. Blank lines may be inserted for readability.

```
EXPROC = eeeeeeee
INCLUDE = iiiiiii
RECORD = rrrrrrr

any number of intermixed LITERAL and FIELD commands

CURSOR = x,y
END
```

MDF Commands

EXPROC - the name of the external procedure to be generated by CRTMAP

INCLUDE - the name of the %INCLUDE file which contains the TYPE declaration of the record to be displayed and all TYPEs and CONSTants to which it refers

example:
```
INCLUDE = TYPES.DCL
```

RECORD - the name of the record data type to be displayed - this type declaration is in the include file

LITERAL - causes a character string to be displayed on the CRT screen, the string must be entered between single quotes

```
LITERAL column, row, 'literal string to be displayed'
```

examples:
```
LITERAL Ø,Ø,'* this is the upper left corner'
```

```
LITERAL 40,12,'* this is about the center'
LITERAL 0,23,'bottom row of the crt'
```

screen coordinates have the origin 0,0 in the upper left corner, first number X is the column, second number Y is the row

FIELD - causes a field in the input record to be displayed at the specified location, may include optional minimum width and decimal places numbers for integers and reals

```
FIELD column, row, field_name {:min_width {:dec_places}}
```

examples:

```
FIELD 10,20, customer_name
FIELD 12,20, account_balance:10:2
FIELD 20,60, days_until_armageddon:1
```

CURSOR - specifies where the cursor should be positioned on the screen after the record is displayed

```
CURSOR column, row
```

END - indicates end of Map Description File, ALWAYS required

15.3 Operating CRTMAP

To operate CRTMAP, first prepare the Map Description File (section 15.2). Prepare a file containing the record to be displayed and its subordinate type declarations - this will be the INCLUDE file.

Make sure the CRTMAP utility was modified to support your terminal type (see section 15.).

To run the utility enter:

```
EXEC CRTMAP
```

It will ask for the "filename.type" of your Map Description File.

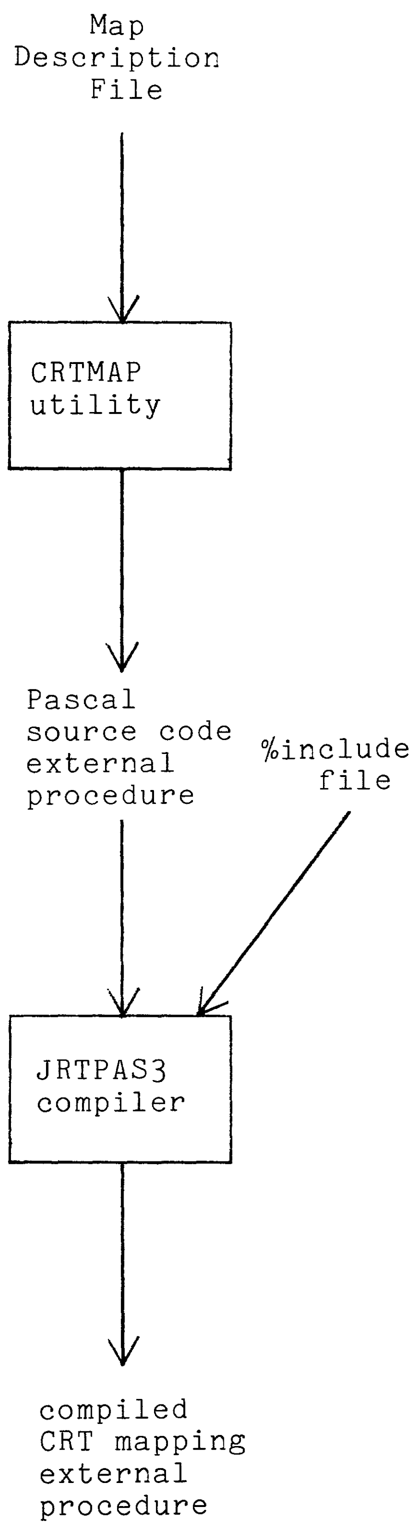
On successful termination of CRTMAP, the new external procedure source file will be found on the default disk. It must be compiled with the JRT Pascal version 3 compiler.

15.4 CRTMAP example

An example of the use of the CRTMAP utility is provided here. A simple customer record is formatted and displayed. The Map Definition File named MDF is listed. The include file named CUSTOMER.PAS contains the main record declaration CUST and a subordinate declaration CHAR30.

The external procedure generated by CRTMAP is named CUSTMAP.PAS and is listed.

A complete compiler listing of CRTMAP.PAS follows.


Map
Description
File
CRTMAP
utility
Pascal
source code
external
procedure
%include
file
JRTPAS3
compiler
compiled
CRT mapping
external
procedure

CRT Screen formatted by CUSTMAP external procedure

```
---------- CUSTOMER RECORD ----------

     Name  PASCAL, BLAISE

     Addr  777 RUE D'ARGENT

     City  PARIS



     Balance $  1490.34
```

File CUSTOMER.PAS
comtains TYPE declaration of customer data record

```
CHAR30 = ARRAY [1..30] OF CHAR;
CUST =  RECORD
        NAME : CHAR30;
        ADDRESS : CHAR30;
        CITY : CHAR30;
        BALANCE : REAL;
        END;
```

FILE MDF
contains Map Definition File which describes CRT screen format

```
EXPROC = CUSTMAP
INCLUDE = CUSTOMER.PAS
RECORD = CUST

LITERAL = 0,0,'---------- CUSTOMER RECORD ----------'

LITERAL = 5,3,'Name '
FIELD = 12,3,NAME

LITERAL = 5,5,'Addr '
FIELD = 12,5,ADDRESS

LITERAL = 5,7,'City '
FIELD = 12,7,CITY

LITERAL = 5,14,'Balance $'
FIELD = 15,14,BALANCE:8:2

CURSOR = 0,22
END
```

```
{ CRTMAP generated external procedure }
extern

type
%include ('CUSTOMER.PAS   '')

procedure CUSTMAP         ( var r : CUST               );

procedure clear;
begin
write(chr(27),'*');
end;

procedure gotoxy ( x,y : integer );
begin
write(chr(27),'=',chr(y+20h),chr(x+20h));
end;

procedure display;
begin
clear;
gotoxy( 0             ,0              );
write('---------- CUSTOMER RECORD ----------');
gotoxy( 5             ,3              );
write('Name ');
gotoxy( 12            ,3              );
write( r.NAME          );
gotoxy( 5             ,5              );
write('Addr ');
gotoxy( 12            ,5              );
write( r.ADDRESS       );
gotoxy( 5             ,7              );
write('City ');
gotoxy( 12            ,7              );
write( r.CITY          );
gotoxy( 5             ,14             );
write('Balance $');
gotoxy( 15            ,14             );
write( r.BALANCE:8:2   );
gotoxy( 0             ,22             );
end;

begin
display;
end;.
```

```
                ----- CRT Mapping Utility -----

0000    %page(50)
0000
0000    { This version setup for Televideo terminals.  To adapt t
0000    terminals modify PROCEDURE PART2 which generates the curs
0000    positioning (gotoxy) and clear screen (clear) codes. }
0000
0000    program crtmap;
0000
0003    type
0010    char16 = array [1..16] of char;
0010
0010    var
0010    ch : char;
0010    alphameric : set of char;
0010    end_of_file : boolean;
0010    map_file_name : string[15];
0010    word : char16;
0010    exproc_name : char16;
0010    include_name : char16;
0010    record_name : char16;
0010    f1, f2 : file of char;
0010
0010
0010    procedure error ( msg : string[40] );
0013    var
0013    dummy : char16;
0016    begin
001A    writeln;
001E    writeln;
0028    writeln(msg);
002C    writeln;
002C    { abnormally terminate - return to CP/M }
0034    call(0,dummy,dummy);
0035    end;
0035
0035    procedure get_char;
003B    begin
004C    read(f1; ch);
0081    if ch = chr(1ah) then error('Premature end of input file'
008D    write(ch);
008E    end;
008E
008E    procedure get_word;
0091    label 99;
0091    var
0091    i : integer;
0094    begin
009D    word := ' ';
00AC    while not (ch in alphameric) do
00AC            begin
```

```
00B1                get_char;
00B4                end;
00C4        word[1] := ch;
00C9        i := 2;
00CE        get_char;
00DC        while (ch in alphameric) do
00DC                begin
00EF                word[i] := ch;
00F9                i := i + 1;
00FE                get_char;
0101                end;
010E        word := upcase(word);
010F        end; {get_word}
010F
010F
010F        procedure init;
0115        begin
012C        writeln('CRTMAP  ver 3.0');
0130        writeln;
0157        write('name of Map Description File : ');
0160        readln(map_file_name);
0164        writeln;
0168        writeln;
0177        reset(f1,map_file_name,binary,256);
017C        end_of_file := false;
0185        ch := ' ';
01A7        alphameric := ['A'..'Z','a'..'z','0'..'9',':','.'];
01AC        get_word;
01E1        if word <> 'EXPROC' then error('EXPROC command expected');
01E6        get_word;
01F2        exproc_name := word;
020A        rewrite(f2, exproc_name + '.pas', binary, 256);
020F        get_word;
0246        if word <> 'INCLUDE' then error('INCLUDE command expected');
024B        get_word;
0257        include_name := word;
025C        get_word;
0291        if word <> 'RECORD' then error('RECORD command expected');
0296        get_word;
02A2        record_name := word;
02A3        end; {init}
02A3
02A3
02A3        procedure part1;
02A9        begin
02DF        writeln(f2; '{ CRTMAP generated external procedure }');
02F4        writeln(f2; 'extern');
02FF        writeln(f2);
0312        writeln(f2; 'type');
033C        writeln(f2; '%include (''',include_name,''')');
```

```
0347    writeln(f2);
0386    writeln(f2; 'procedure ',exproc_name, '( var r : ', recor
');');
0391    writeln(f2);
0392    end; {part1}
0392
0392
0392    procedure part2;
0398    begin
03B7    writeln(f2; 'procedure clear;');
03CB    writeln(f2; 'begin');
03ED    writeln(f2; 'write(chr(27),''*'');');
0400    writeln(f2; 'end;');
040B    writeln(f2);
043D    writeln(f2; 'procedure gotoxy ( x,y : integer );');
0451    writeln(f2; 'begin');
0489    writeln(f2; 'write(chr(27),''='',chr(y+20h),chr(x+20h));'
049C    writeln(f2; 'end;');
04A7    writeln(f2);
04A8    end; {part2}
04A8
04A8
04A8    procedure part3;        {create DISPLAY procedure}
04A8
04AB    procedure process_coordinates;
04AE    var
04AE    x_coord, y_coord : char16;
04B1    begin
04B6    get_word;
04C2    x_coord := word;
04C7    get_word;
04D3    y_coord := word;
0507    writeln(f2; 'gotoxy( ',x_coord,',',y_coord,');');
0508    end;
0508
0508    procedure process_string;
050E    begin
050E    {find start of string}
052E    while not (ch in ['''',chr(0dh),' ',chr(9),chr(1ah)]) do
0536            get_char;
0566    if ch <> '''' then error('Literal string expected');
057B    write(f2; 'write(');
057B    repeat
058E      write(f2; ch);
0593      get_char;
05A1    until ch = chr(0dh);
05B2    writeln(f2; ');');
05B3    end;
05B3
05B3
05B6    begin {part3}
```

```
05D7     writeln(f2; 'procedure display;');
05EB     writeln(f2; 'begin');
0600     writeln(f2; 'clear;');
0608     while not end_of_file do
0608             begin
060D             get_word;
0613             case word of
0621             'LITERAL' :
0621                     begin
0626                     process_coordinates;
062B                     process_string;
062E                     end;
063A             'FIELD' :
063A                     begin
063F                     process_coordinates;
0644                     get_word;
066C                     writeln(f2; 'write( r,',word,');');
066F                     end;
0684             'CURSOR' : process_coordinates;
0696             'END' : end_of_file := true;
06D3             else : error('LITERAL, FIELD, CURSOR or END command
cted');
06D4             end;
06D7             end;
06EA     writeln(f2; 'end;' );
06F5     writeln(f2);
06F6     end; {part3}
06F6
06F6
06F6     procedure part9;
06FC     begin
0710     writeln(f2; 'begin');
0727     writeln(f2; 'display;');
073B     writeln(f2; 'end;.');
073C     end; {part9}
073C
073C
073F     begin {crtmap}
0744     init;
0749     part1;
074E     part2;
0753     part3;
0758     part9;
075C     close(f1);
0760     close(f2);
0761     end {crtmap}.
No errors detected
Module size = 1893 dec bytes
End of compile for CRTMAP
```

A. Reserved words

The following words are reserved in JRT Pascal and may not be used as identifiers.

abs
addr
allocate
and
array
begin
binary
boolean
call
case
char
chr
close
compress
concat
cons
const
copy
deallocate
delete
dispose
div
do
downto
else
end
eof
eoln
extern
false
file
fillchar
for
forward
free
function
get
goto
hex$
if
in
include

initialize
input
insert
integer
label
length
list
lrange
ltrace
map
maxint
mod
new
nil
nocons
nolist
noltrace
noptrace
not
nowarning
odd
of
open
or
ord
output
page
portin
portout
pos
pred
procedure
program
ptrace
put
rba
read
readln
real
real$
record
repeat
reset
rewrite
round
rrn
set
sqr

succ
string
system
text
then
title
to
true
trunc
type
until
upcase
var
warning
while
with
write
writeln
xor

B. Activity analyzer

The activity analyzer - Activan - is a facility which moniters the execution of a Pascal program and prints a graph showing the amount of time spent executing each portion of the program. To use Activan, a program must be compiled with the %LTRACE directive or the $L compile switch on.

Activan moniters the line numbers as a program executes and keeps counters for the line numbers in the specified range. The range of line numbers to be monitered and the line spacing can be set and changed when the program is running.

To run a program with Activan, specify the $A switch when the program is started with the EXEC command.

```
EXEC TESTPGM  $A
```

Before the program begins execution Activan will request console input to specify the line range to be monitored and the line spacing. When those parameters have been entered, program execution will begin.

If Activan is active when the program terminates, Activan mode is entered so that a final histogram can be printed.

While the program in running, it can be interrupted and control returned to Activan by keying in a control-A character. Activan will then request which action is desired:

code	action
C	clear the counters to zero
E	end the program
H	print histogram of activity
I	initialize the line range and spacing
R	run the program with Activan monitoring
W	run the program without Activan

C. Block letters

An external procedure named LETTERS is provided to generate large block letters. These letters are 9 lines high and from 4 to 1Ø columns wide. The external procedure generates an entire row at a time of letters for use as report headers, program identifiers, etc. The output line may be up to 22Ø columns wide.

The upper case letters, numbers, and dash may be input to the external procedure. Unsupported characters are converted to spaces. Lower case characters are converted to upper case.

The output from LETTERS is placed in a buffer which is an array of strings - this must be defined exactly as shown. The declaration for LETTERS is:

```
TYPE
BUFFER = ARRAY [1..9] OF STRING[22Ø];

PROCEDURE LETTERS ( INPUT_STRING : STRING;
                    SLANT : CHAR;
                    VAR B : BUFFER ); EXTERN;
```

The input_string is the line of characters to be converted to block letter format. The slant character provides for 'streamlined' characters by slanting left or right. Slant may be 'L' or 'R' or ' '. The output buffer b refers to a variable of type buffer in the users program. Note that b is a reference parameter.

This sample program will print out the word 'PASCAL' in block letters.

```
PROGRAM BLOCKS;

TYPE
BUFFER = ARRAY [1..9] OF STRING[220];

VAR
I : INTEGER;
BLOCKS_BUFR : BUFFER;

PROCEDURE LETTERS ( INPUT_STRING : STRING;
                    SLANT : CHAR;
                    VAR B : BUFFER ); EXTERN;

BEGIN
LETTERS('PASCAL','R',BLOCKS_BUFR);
SYSTEM(LIST);
FOR I:=1 TO 9 DO WRITELN( BLOCKS_BUFR[I] );
END.
```

D. JSTAT

Jstat is an external procedure which can be used to compute several basic statistics given an array of real numbers as input. It computes the arithemetic mean, standard deviation, variance, skewness, kurtosis and the first four moments about the mean.

The source code for jstat is provided on the source disk and may be modified. The procedure is restricted to an array of 1000 real numbers but this can be easily changed by modifying the declaration of the data type jstat_array and recompiling.

While jstat_array is declared as a 1000 element array, a much smaller array may be used to hold the data values since the input array is used as a reference parameter.

Jstat requires three parameters:

n - number of data items in the input array

x - array of up to 1000 real numbers

r - output record containing computed statistics

The following type declarations and procedure declaration are required in the calling Pascal program.

```
TYPE
JSTAT_INTERFACE =
        RECORD
        MEAN, STANDARD_DEVIATION,
        VARIANCE, SKEWNESS, KURTOSIS,
        M1, M2, M3, M4 : REAL;
        END;
JSTAT_ARRAY = ARRAY [1..1000] OF REAL;

PROCEDURE JSTAT ( N : INTEGER;
                  VAR X : JSTAT_ARRAY;
                  VAR R : JSTAT_INTERFACE );
                EXTERN;
```

E. JGRAF

JGRAF is an external procedure which formats x-y graphs and scatter graphs. The graph size in rows and columns and the lower and upper x and y bounds are set by the calling program. A title to the graph may be provided. Once the graph has been prepared, it can be displayed on the console, printed, or stored in a disk file.

Any number of data points can be plotted. Any number of separate plots can be prepared simultaneously (within memory limitations).

To use JGRAF, your program (or occasionally an external procedure) must declare the char9000 and jgraf_interface types. Your program must then declare one (or more) variables of type jgraf_interface. For convenience, the interface variable will be called jgi in this document. Your program could call the interface variable(s) anything appropriate. Your program must also declare JGRAF as an external procedure.

The declarations for sample main program to take plotting commands from a disk file and create a plot is shown here. (The body of the sample program is listed later.) Everything listed here is required of any program using JGRAF except for the declarations noted as specific to jg.

```
program jg;
%ltrace %ptrace  (* optional - suggested *)

type
char9000 = array [1..9000] of char;
jgraf_interface = record
        command : char;           (* R *)
        plot_char : char;         (* R *)
        x_grid : boolean;         (* R *)
        y_grid : boolean;         (* R *)
        rows : integer;           (* R *)
        columns : integer;        (* R *)
        x_lower : real;           (* R *)
        x_upper : real;           (* R *)
        y_lower : real;           (* R *)
        y_upper : real;           (* R *)
        filename : array [1..14] of char;
        title : string;           (* R *)

(* fields below used internally by jgraf *)
        b : ^char9000;
        bufr_size : integer;
        line_size : integer;
        row_count : integer;
        x_spacing : real;
        y_spacing : real;
        end;

var
jgi : jgraf_interface;

(* following are used by program jg *)
file_name : array[1..20] of char;
title : array[1..24] of char;
inf : file of char;
x, y : real;
command : char;
(* end of variables used by sample program *)

procedure jgraf ( var jg : jgraf_interface;
                  x, y : real ); extern;

(* end of declarations *)
```

To produce graphs, your program must first set all members of jgi marked (* R *) in the jgraf_interface type declaration to appropriate values.

Jgi.x_grid would be set to false if grid lines running across the graph should be omitted. Jgi.y_grid is set to false if grid lines running up and down are to be omitted. Jgi.rows and jgi.columns contain the number of lines and number of characters across the body of the plot itself (minus one).

The number of rows and columns should normally be divisible by 10. Plot size can be calculated as (number of columns + 16) * (number of lines + 5), which should not exceed 9000 characters. The length of jgi.title should be less than the number of columns in the plot.

Once all the required members of jgi are initialized, set jgi.command to 'I' and call JGRAF, as

```
JGI.COMMAND := 'I';
JGRAF ( JGI, 0.0, 0.0 );
```

(Note that the examples listed here in upper case are for illustration only and are not part of the program jg.)

Then, to place data points on the graph, set jgi.command to 'D' and call JGRAF for each of the appropriate points. Do this as often as needed. To get two distinct curves, you could get jgi.plot_char to '*' for one set of points, then set it to '#' before calling JGRAF with another set of points.

```
JGI.COMMAND := 'D';
JGI.PLOT_CHAR := '*';
JGRAF ( JGI, 15.4, 199.2 );
JGRAF ( JGI, 15.9, 205.7 );
JGI.PLOT_CHAR := '#';
JGRAF ( JGI, 9.0, 105.0 );
```

To print the graph on the console, set jgi.command to 'C' and call JGRAPH with x and y arguments zero, as

```
JGI.COMMAND := 'C';
JGRAPH ( JGI, 0.0, 0.0 );
```

If you want output to the line printer as well as the console, set jgi.command to 'P' instead of 'C' before calling JGRAF.

To write the graph to a file, set jgi.filename to the desired name, jgi.command to 'S', and call JGRAF.

```
JGI.FILENAME := 'B:PLOT.5';
JGI.COMMAND := 'S';
JGRAF( JGI, 0.0, 0.0 );
```

More data points can be added to a graph after printing, so that development or trends can be plotted in succession. Further, by setting jgi.plot_char to a space (' '), data points can be erased (though the grid lines will not be restored).

If you want to print more than one graph using the same interface record (jgi) or want JGRAF to free the memory allocated to produce a graph, you can set jgi.command to 'X' before calling JGRAF. This will free the buffers allocated by JGRAF (in the I command).

Note that every call to JGRAF that is not providing data (jgi.command = 'D') should have the x and y arguments equal to 0.0.

The body of the sample program jg is included here, and illustrates one use of JGRAF. Jg takes a disk file of commands as input and produces one or more plots as directed. Commands on the disk file are similar to the options to JGRAF, with the addition of two commands. T followed by 'title' may preceed the I command. Period (.) followed by a space and a new plot character will reset the current plot character.

```
begin   (* jg *)
write('General graphing input file: ');
readln(file_name);
reset(inf, file_name, text, 512);
jgi.title := ' ';
while (not eof(inf)) do
        begin
        read(inf; command);
        command := upcase(command);
        writeln('db ', command);
        jgi.command := command;
        case command of
        'T':    begin
                readln (inf; title);
                jgi.title := title;
                end;
```

```
                'I':    begin
                        readln (inf; jgi.rows, jgi.columns,
                                jgi.x_lower, jgi.x_upper,
                                jgi.y_lower, jgi.y_upper);
                        jgi.plot_char := '*';
                        jgi.x_grid := true;
                        jgi.y_grid := true;
                             (* note that all required members *)
                             (* of jgi have been set *)
                        jgraf(jgi, Ø.Ø, Ø.Ø);
                        writeln(' done I');
                        end;
                'D':    begin
                        read(inf; x, y);
                        jgraf(jgi, x, y);
                        end;
                '.':    readln(inf; jgi.plot_char);
                'C':    jgraf(jgi, Ø.Ø, Ø.Ø);
                'P':    jgraf(jgi, Ø.Ø, Ø.Ø);
                'S':    begin
                        readln(inf; file_name);
                        jgi.filename := file_name;
                        jgraf(jgi, Ø.Ø, Ø.Ø);
                        end;
                'X':    jgraf(jgi, Ø.Ø, Ø.Ø);
                else:   writeln('Unrecognized command: ', command);
                        end;
                end;
        close(inf);
        end.
```

Given the input file SAMPLE.DAT as follows

```
        T 'Sample'
        I 2Ø 4Ø Ø 4Ø Ø 6Ø
        D 5 6   D 6 1Ø
        D 7 12  D 8 15
        D 9 16  D 1Ø 16
        . #
        D 5 2   D 32 6
        D 32 27
        C
        S sample.out
        X
```

Jg will produce the (uninspired) output file SAMPLE.OUT as follows given the input listed above.

```
JGRAF ver 3.0    **** Sample                ****

         60 -I---------I---------I---------I---------I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
         30 -I---------I---------I---------I---------I
             I         I         I         I #       I
             I         I         I         I         I
             I         I         I         I         I
             I         I         I         I         I
             I       ***         I         I         I
             I      *  I         I         I         I
             I     *   I         I         I         I
             I    *    I         I         I #       I
             I    #    I         I         I         I
          0 -I---------I---------I---------I---------I

             0         10        20        30        40
```

A summary of the commands to JGRAF is included for reference:

code	meaning
C	display graph on console
D	plot a data point
I	initialize graph buffer and axes
P	print graph
S	save graph on a disk file
X	delete graph buffer

The source code for jgraf is provided and may be modified. For example, the number of lines between the x_grid lines can be changed to 6 (or 8) so that grid lines form a one inch square on printers with 10 characters per inch and 6 (or 8) lines per inch.

JGRAF is not limited to scatter plots. With appropriate selection of data points, histograms can be produced. Contour plots (and even isometric drawings) are also possible.

F. Restrictions

1. Arrays are limited to 8 dimensions.

2. Literal character strings in the "const" section are limited to 32 characters.

3. Random disk files require CP/M 2.2 and may be up to 8 megabytes in size.

4. Sets are limited to 128 elements. The first element (leftmost) corresponds to 0, the last (rightmost) corresponds to 127.

5. Not more than 63 external procedures and functions may be declared.

6. Not more than 1632 dynamic storage blocks may be allocated at one time. The run-time system may require up to 100 of these for file buffers, file control blocks, external procedures and other uses.

7. "With" statements may not be nested to more than 31 levels.

8. "Case" statements are limited to 128 clauses and 128 labels per clause.

9. Integers must be between +32767 and -32768, since they are stored in 16 bit twos complement format. In a few cases integers will be treated as unsigned 16 bit values with a range of 0 to +65535. The MAP and CALL builtin procedures require addresses which may range up to 65535. Accessing random files by relative byte address may require byte addresses up to 65535.

10. "Real" numbers are represented in 14 digit binary coded decimal format. The floating point exponent range is from -64 to +63.

11. The names of procedures and functions may not be used as parameters.

12. Literal character strings in the source program may not exceed 127 characters.

P R O B L E M   R E P O R T

MAIL TO: JRT Systems
Technical Services
45 Camino Alto
Mill Valley, CA 94941

Name ______________________________

Company ___________________________

Address ___________________________

City ______________________ State __________ Zip _____

CP/M version ___________ Disk format _______________

Date ___________________ Approx purchase date _________________

Please include as much information as possible about the problem. A listing of the program code is essential for us to duplicate the problem.

Did problem occur during compile? ____
execution ____ linker ____ assembly ____
other ________________

Was there an error message? Which one?

Complete description of problem:

Are symptoms always the same or do they vary?