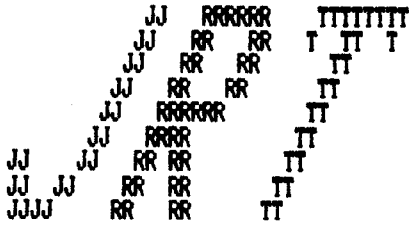
JRT SYSTEMS

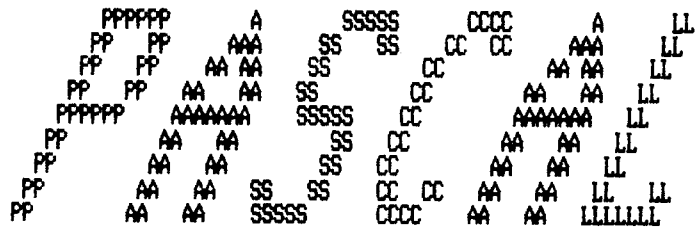
PO Box 22365

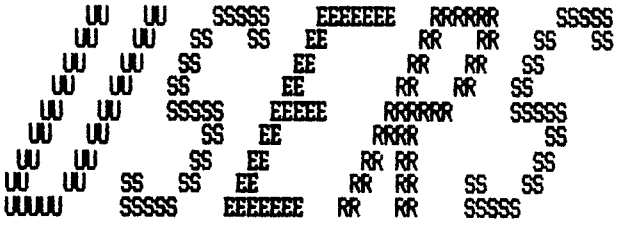
San Francisco

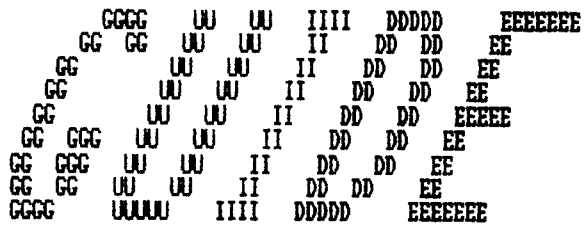
CA 94122

PASCAL


JJ RRRRRR TTTTTTTT
JJ RR RR T TT T
JJ RR RR TT
JJ RR RR TT
JJ RRRRRR TT
JJ RRRR TT
JJ JJ RR RR TT
JJ JJ RR RR TT
JJJJ RR RR TT


PPPPPP A SSSSS CCCC A LL
PP PP AAA SS SS CC CC AAA LL
PP PP AA AA SS CC AA AA LL
PP PP AA AA SS CC AA AA LL
PPPPPP AAAAAAA SSSSS CC AAAAAAA LL
PP AA AA SS CC AA AA LL
PP AA AA SS CC AA AA LL
PP AA AA SS SS CC CC AA AA LL LL
PP AA AA SSSSS CCCC AA AA LLLLLLL


UU UU SSSSS EEEEEEE RRRRRR SSSSS
UU UU SS SS EE RR RR SS SS
UU UU SS EE RR RR SS
UU UU SS EE RR RR SS
UU UU SSSSS EEEEE RRRRRR SSSSS
UU UU SS EE RRRR SS
UU UU SS EE RR RR SS
UU UU SS SS EE RR RR SS SS
UUUUU SSSSS EEEEEEE RR RR SSSSS


GGGG UU UU IIII DDDDD EEEEEEE
GG GG UU UU II DD DD EE
GG UU UU II DD DD EE
GG UU UU II DD DD EE
GG UU UU II DD DD EEEEE
GG GGG UU UU II DD DD EE
GG GGG UU UU II DD DD EE
GG GG UU UU II DD DD EE
GGGG UUUUU IIII DDDDD EEEEEEE

COPYRIGHT

DISCLAIMER

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

TRADEMARKS

JRT Pascal is a trademark of JRT Systems. CP/M is a registered trademark and MP/M is a trademark of Digital Research.

JRT Pascal User's Guide

Table of Contents

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

```
JRTPAS2.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
```

System customization program

```
CUSTOMIZ.INT
```

Block letters external procedure

```
LETTERS.INT
```

Dynamic trace control external procedure

DEBUG.INT

Utility to convert Microsoft modules
CONVERTM.INT

Statistics external procedure
JSTAT.PAS
JSTAT.INT

Graph preparation external procedure
JGRAF.PAS
JGRAF.INT

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

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             0EADH
12FH            0cf00h
-0ffffh         +50h
```

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         0.000098
250000.000321   0.442e+35
2.0E-60         -15.011e+03
```

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:

```
JRTPAS2  filename  <$ options>
```

Examples:

```
JRTPAS2 TESTPGM

JRTPAS2 STATISTC $E

JRTPAS2 INVENTRY $ELP

C:JRTPAS2  B:PROJECT1 $E

JRTPAS2 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

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:

JRTPAS2.COM	21K
PASCAL0.INT	21K
PASCAL1.INT	7K
PASCAL2.INT	5K

```
PASCAL3.INT      9K
PASCAL4.INT      1K
PASCAL.LIB      13K
```

The compiler does not need to be located on the A: disk. The main compiler module JRTPAS2.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	issue a form feed character to start a new page

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.

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'.

32.01e+04 1.075 -3.14159 -1234567.8901234E-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[1500] );
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  := ['0'..'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.

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	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 ));

    .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(0FFH) DO  I:=I+1;

TAB := CHR( 9 );

CARRIAGE_RETURN := CHR( 0DH );

LINE_FEED := CHR( 0AH );
```

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( source_string, 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 >= 0 THEN
        REALFREE := TEMP
ELSE
        REALFREE := 65536.0 + TEMP;
END;
```

Examples:

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

IF REALFREE <= 2000.0 THEN
        WRITELN('STORAGE CRITICAL');

IF FREE >= 1500 THEN NEW( BUFFER );

IF FREE >= 4096 THEN BUFSIZE:=2048
ELSE BUFSIZE:=1024;
RESET( INFILE, 'TEST.DAT', BINARY, BUFSIZE );
```

5.10 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 8080/Z80 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(string_expression);

The LENGTH function returns an integer value which is the current length of the string variable or expression. It can be used with dynamic strings or structured variables.

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+1.0;

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 : 0
TEST 3 : 3
TEST 4 : 5
```

5.17 PRED

Format 1
```
PRED( integer_expression );
```

Format 2
```
PRED( character_expression );
```

The PRED function returns the 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:

```
' +0.12345678901234E+00'
```

Examples:

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

STR := REAL$( VELOCITY / 7.03E-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 0.5 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 );
```

5.20 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 ));
```

5.21 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 );
```

5.22 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 );
```

5.23 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) );
```

5.24 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 );
```

5.25 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);

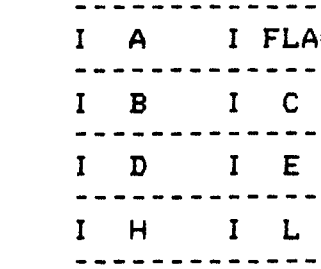
The CALL builtin procedure allows you to make direct calls to the CP/M operating system, to your own Basic Input/Output System (BIOS), and to any machine language code present in main storage. The 8080 data registers can be directly setup for passing parameters to the module called. The 8080 data registers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



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 0001 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
130	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
140	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
150	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 01 THEN THE
OPERATING SYSTEM IS MP/M                  *)
CASE ORD( RETURNED_REGS.H ) OF
0 : WRITE('CP/M ');
1 : WRITE('MP/M ');
ELSE : WRITE('????');
END;
WRITE(' VERSION ');

CASE HEX$( RETURNED_REGS.L ) OF
'00' : WRITELN('1.X');
'20' : WRITELN('2.0');
'22' : WRITELN('2.2');
ELSE : WRITELN( HEX$( 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;
        R0, R1, R2 : CHAR;
        END;
VAR
FCB : FILE_CONTROL_BLOCK;
PARM_REGS, RETURNED_REGS : DATA_REGISTERS;

BEGIN
(* SET UP FCB *)
FCB.DISK := CHR(0);
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. 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 0 to 0FFFFH.

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 80 VIDEO TERMINAL      *)
(* IT IS A MEMORY-MAPPED MODEL WITH ITS *)
(* VIDEO SCREEN BEGINNING AT 0F000H     *)

TYPE
SCREEN = ARRAY [1..24, 1..80] OF CHAR;
VAR
CRT : ^SCREEN;
BEGIN
MAP( CRT, 0F000H );

(* 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 0 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. Three modes of input/output - console, sequential disk, random 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.

The procedures GET and PUT are not supported. The standard procedures READ and WRITE are extended to support every processing need.

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.

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
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

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 = 0. 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 = 0.

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
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',MTD_SALES:9:2,
        '   YTD sales',YTD_SALES:9:2);
  WRITELN;
10: (* EXIT LABEL *)
UNTIL DEPT = 999;

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

7.4 Indexed file processing

In most applications where random or direct file access is needed, there will not be a one-for-one match between the key and the relative record number. In these cases some form of index must be used to match the key to the record number or relative byte address of the desired data item in the file.

The index itself may be located in the file and be maintained as the file changes. It must contain at least a key and a data location field for each record.

The key which is used to locate the data is usually some value like department number, customer name or supplier number concatenated to part number. If the key itself is large then the index could become very large and occupy too much main storage. In this case a shorter key can be created from the original key data. For example, a four byte key could be generated from the first, third, eighth and tenth letters of a customer name. Duplicate keys can occasionally occur and may be considered in programming the index search procedure.

When the file contains a very large number of records or data items a two level index may be used. The primary index which is kept in storage contains the range of keys contained in each of the second level indexes. The primary index is searched for the correct key range. The correct second level index is loaded and searched. Finally the actual record is loaded. In many applications the one extra disk access would be justified by the savings in storage.

For the experienced programmer, the POS builtin string function can be used to perform very fast searches of indexes.

One method of indexed file processing places the index as the first record on the file. The index will contain a key in any useful format, an RBA value, and perhaps a record size field for variable records.

After opening this indexed file, the index is read from RBA=0 into an array in storage. There it can be searched for any particular key. If the record is found then using the RBA from the index it can be loaded into storage. It can be updated if necessary and rewritten.

A sample program segment based on this kind of indexed file is shown. It provides online access to a file of message texts. The indexed file could be created by a separate sequential disk program.

```
(*** GLOBAL TYPE AND VAR DECLARATIONS ***)
CONST
INDEX_SIZE = 100;
TYPE
INDEX_ENTRY = RECORD
              MSG_NUM   : INTEGER;
              MSG_RBA   : INTEGER;
              END;
INDEX = ARRAY [1..INDEX_SIZE] OF INDEX_ENTRY;

VAR
IX : INDEX;
MSG_FILE : FILE OF CHAR;


PROCEDURE MESSAGE ( NUM : INTEGER );
VAR
I : INTEGER;
MSG_BUFFER : ARRAY [1..1000] OF CHAR;
BEGIN
IF NUM = 0 THEN  (* INITIALIZE *)
        BEGIN
        OPEN( MSG_FILE, 'B:MESSAGE.DAT',
                BINARY );
        READ( MSG_FILE, RBA, 0; IX);
        END
ELSE
        BEGIN   (* LOCATE AND PRINT MSG *)
        I:=1;
        WHILE (I <= INDEX_SIZE)
                AND (NUM <> IX[I].MSG_NUM) DO
                I:=I+1;
        IF I = INDEX_SIZE THEN
                WRITELN('Unknown message',NUM)
        ELSE  (* LOAD MESSAGE *)
                BEGIN
                READ( MSG_FILE, RBA,
                   IX[I].MSG_RBA; MSG_BUFFER);
                WRITELN( MSG_BUFFER );
                END;
        END;
END; (* MESSAGE *)
```

7.5 EOF

Format
EOF (filename);

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 true. 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 (filename);

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 := 0;  COUNT := 0;
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 := 0;
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 OPEN

Format 1
```
OPEN ( file_identifier, filename, BINARY );
```

Format 2
```
OPEN ( file_identifier, 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_identifier 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 );
```

7.9 READ, READLN

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

Format 2  (sequential disk)
READ/LN ( file_identifier ; variable1, variable2,... );

Format 3  (random disk)
READ/LN ( file_identifier, RRN, integer_or_real_expr ;
                variable1, variable2,...);

Format 4  (random disk)
READ/LN ( file_identifier, 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_identifier 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_identifier 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 *)
```

7.10 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' );
```

7.11 RESET

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

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

The RESET standard procedure is used to open already existing files for sequential input. If the file specified is not present, a run-time error occurs.

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

The file_identifier 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 );
```

7.12 REWRITE

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

Format 2
```
REWRITE( file_identifier, 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_identifier 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);
```

7.13 WRITE, WRITELN

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

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

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

Format 4  (random disk)
WRITE/LN ( file_identifier, 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	09h	horizontal tab
10	0ah	line feed
12	0ch	form feed, clear screen
13	0dh	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_identifier 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, 0;  INDEX );

        WRITELN;                    (* BLANK LINE *)

        WRITE( CHR(0CH) );          (* 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 JRTPAS2 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 JRTPAS2. 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:

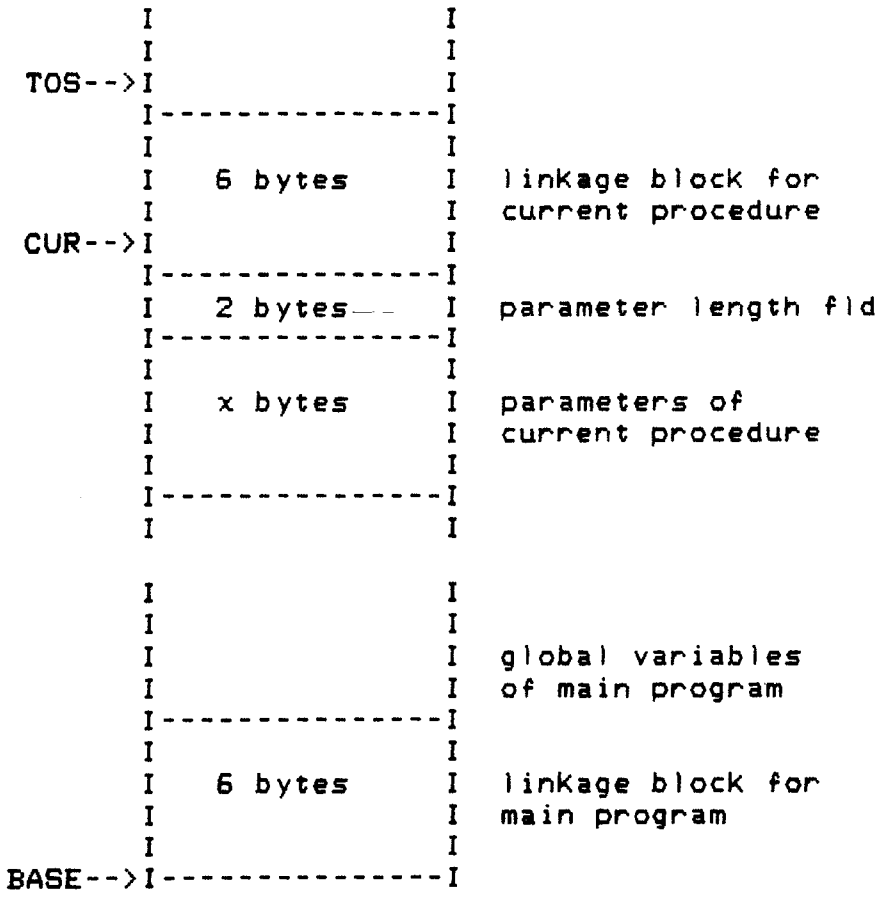
```
*  /  +  -
NOT  AND  OR  XOR
MOD  HIGH  LOW
EQ  NE  LT  LE  GT  GE
```

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
```



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   0700  0300      xx xx xx xx xx xx  yy
                      I                 I
                      CUR               TOS
```

The two byte integer fields are in 8080 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 0300   0800    0400    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 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# 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 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# 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,0feh
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

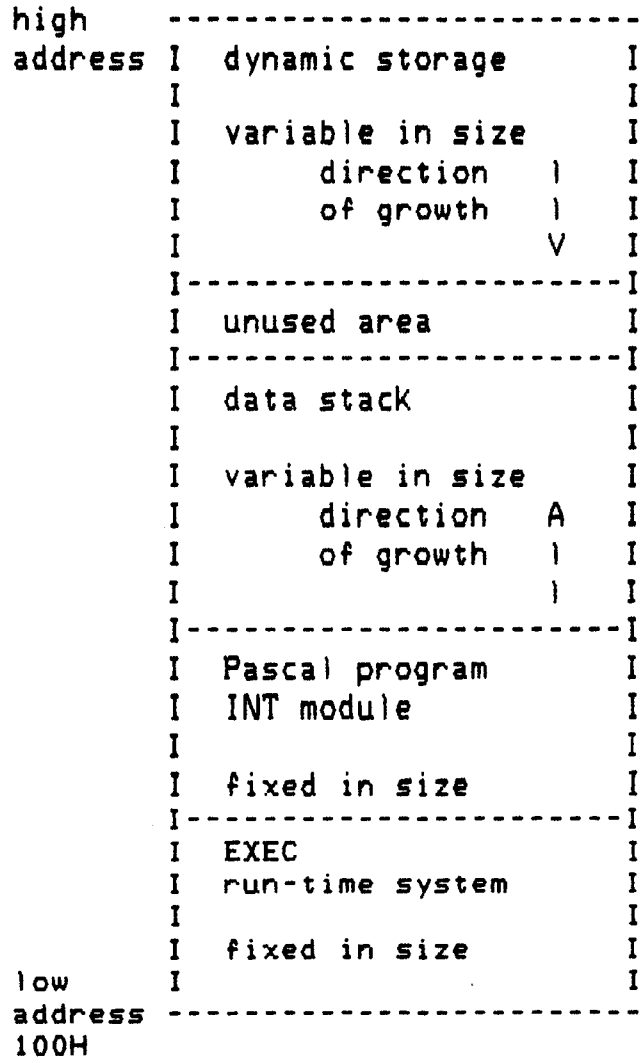
3. Data stack - variable in size, this region begins at the end of the Pascal program and grows toward higher addresses; this region contains all static variables (those created by VAR declarations), parameters passed to procedures and procedure activation blocks

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.



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

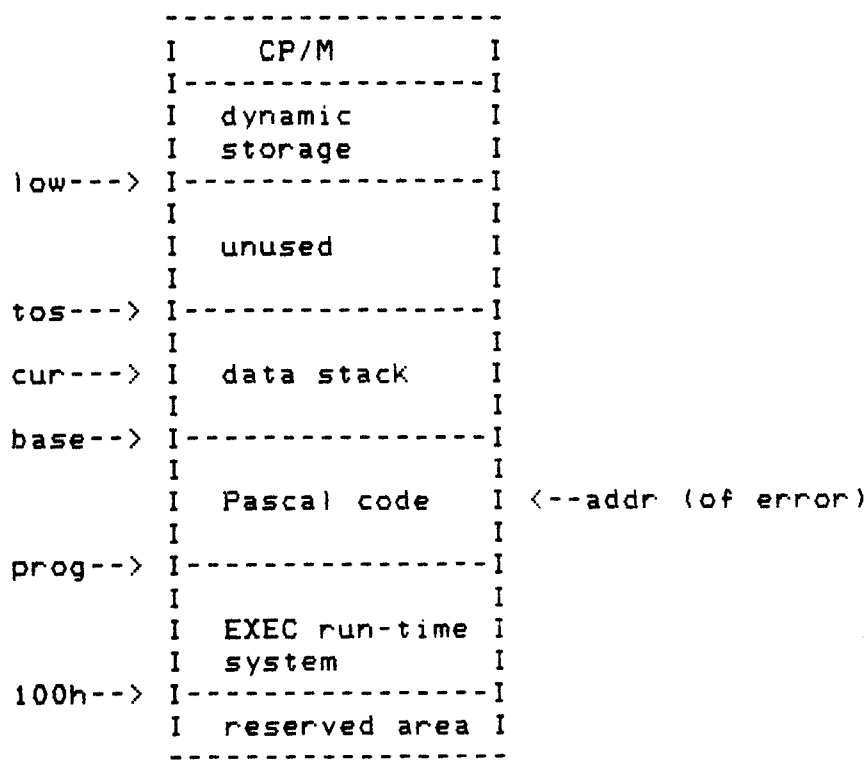
When an error is detected, an error message is displayed on the console. The current line number and last entered procedure name may also be displayed (see section 13.1). A system status display is also created - this contains useful information about the current state of the run-time system.

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.



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
80	file is open
40	random mode - not sequential
20	text mode - not binary
10	EOLN flag set
08	input - not output or random
04	EOF flag set

Formatted external procedure table

exproc name	addr	use cnt	time	stat
ACCTPAY1	C2AE	0000	0004	30
ACCTPAY2	3E22	0000	0165	74
GENLEDG1	0001	0000	0000	00
ACCTREC1	3F55	0001	014E	F4
ACCTREC2	440C	0001	015A	F4
SORT	0001	0000	0000	00
+INVENTRY	503A	0001	020D	F4
CHECKS	5052	0000	0103	30

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 0001 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 0000 (not active) or 0001 (active), it will be greater than 0001 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
80	procedure is currently active
40	procedure was linked with main program
20	procedure is currently in storage
10	procedure file control block is open
04	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

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 compatibile 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');
2.0 * PHI       : WRITELN('TWO PHI');
3.0 * 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 <= 14.08)       : PROCESS_RANGE_2;
(V > 35.6) AND (V <= 100.0)      : PROCESS_RANGE_3;
ELSE : WRITELN('VOLTAGE OUT OF VALID RANGES:',V);
END;
```

A. Reserved words

The following words are reserved in JRT Pascal and may not be used as identifiers.

```
abs
addr
and
array
begin
binary
boolean
call
case
char
chr
close
compress
concat
cons
const
copy
delete
dispose
div
do
downto
else
end
eof
eoln
extern
false
file
fillchar
for
forward
free
function
goto
hex$
if
in
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
rba
read
readln
real
real$
record
repeat
reset
rewrite
round
rrn
set
sqr
succ
string
system
text
then
```

```
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 10 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 220 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[220];

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 in 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 user. 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.

The main interface between a Pascal program and jgraf is a record variable of type jgraf_interface. The setup parameters are stored here and jgraf uses this area for some of its own working variables.

Jgraf performs several different functions, such as initialize, plot data point, save disk file. A command code character in jgraf_interface informs jgraf which operation is required. The first call to jgraf should be the initialize operation 'I'. After that any number of data points may be plotted by setting the command code to 'D' and calling jgraf with the data point (x,y) as parameters. Since the graph is prepared in a buffer in dynamic storage, when graph preparation and display are done, a call with command code 'X' should be used to delete this buffer.

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

All jgraf parameters except the x,y values of data points to be plotted are stored in a record variable jgraf_interface. When calling jgraf the x and y parameters should be zero unless a data point is being plotted (command D). The following declarations are required for use of jgraf.

Declarations required to use JGRAF

```
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
jgraf_file : file of char;

procedure jgraf ( var jgi : jgraf_interface;
                  x, y : real ); extern;
```

IMPORTANT - Jgraf_file is always required and it must be declared as the first file in the main program.

The required parameters in jgraf_interface are flaged here with an R.

The character to be placed on the graph for each data point must be supplied in the parameter plot_char.

Jgraf always plots x and y axes and labels them every ten rows or columns. X and y grids over the entire graph area may optionally be plotted by setting the parameters x_grid and y_grid to true. If grids are not desired these

parameters should be set to false.

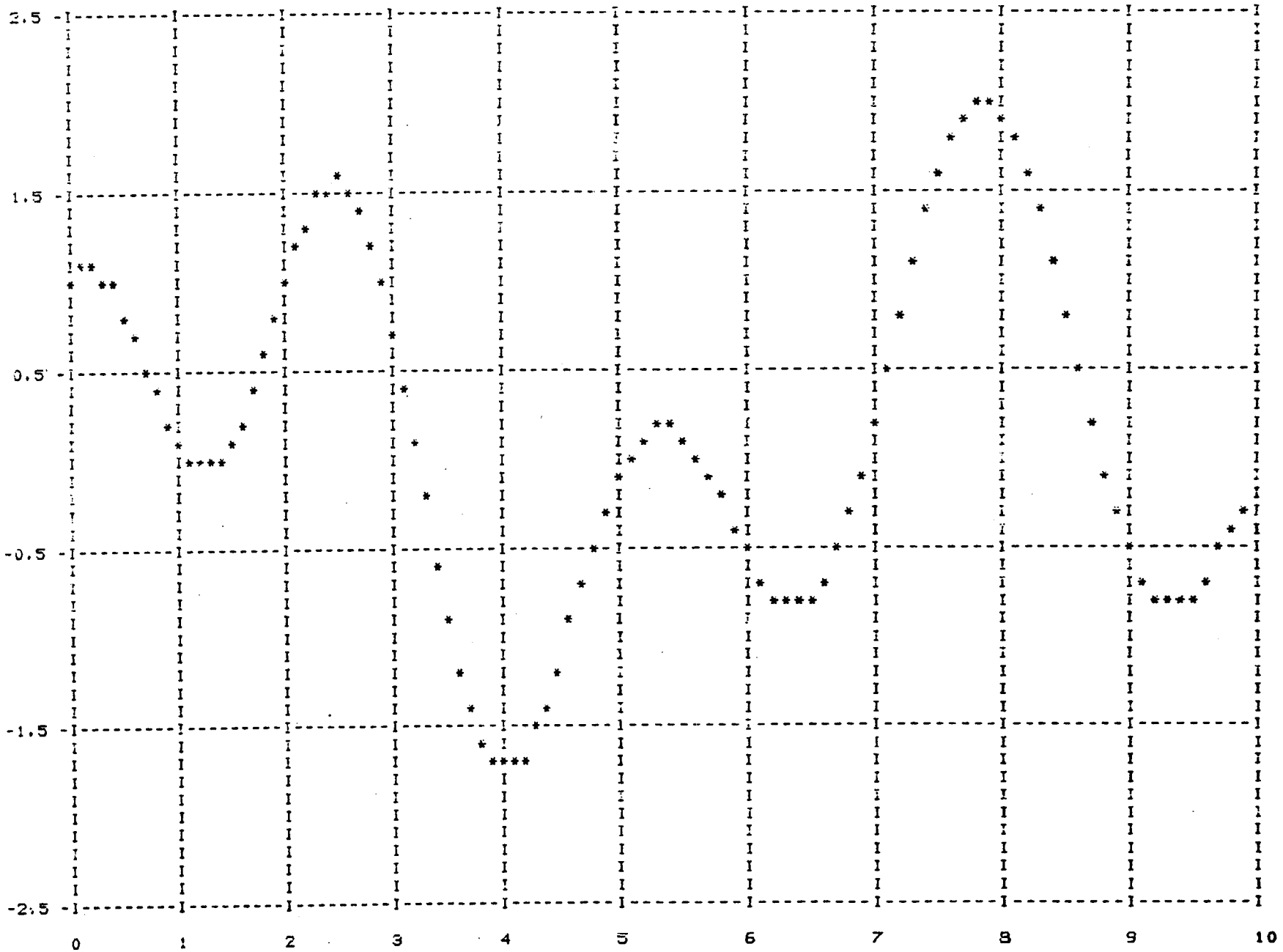
To save the graph on a disk file - store 'S' in the command code and store the disk filename in the filename parameter.

Multiple graphs may be plotted simultaneously by having multiple copies of the jgraf_interface record.

The source code for jgraf is provided and may be modified.

F ver 1.0

**** sin(x) + cos(2.4 * x) ****



F. Restrictions

1. Arrays are limited to 8 dimensions.

2. Not more than 10 files may be declared.

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. File variables may not be used in assignment statements or as parameters.

12. The names of procedures and functions may not be used as parameters.

13. Literal character strings in the source program may not exceed 127 characters.

14. Literal character strings in the "const" section are limited to 32 characters.

15. The functions GET and PUT and buffer variables are not implemented. The standard procedures READ and WRITE are extended to handle any kind of input/ output requirement.

P R O B L E M R E P O R T

MAIL TO: JRT Systems, POB 22365, San Francisco, CA 94122

Name _____

Address _____

City _____ State _____ Zip _____

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?

P R O B L E M R E P O R T

MAIL TO: JRT Systems, POB 22365, San Francisco, CA 94122

Name _____

Address _____

City _____ State _____ Zip _____

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?

JRT PASCAL
OWNER NOTES

November 1982

Your response to our new marketing strategy of very low price/ high quality/ high volume software has exceeded our wildest dreams!

Since May, when we slashed JRT Pascal's price from $295 to $29.95, we've added over 10,000 new customers! -- and we expect to reach 25,000 by year-end!

Because we allow owners to make copies for friends, the actual user number is much larger.

Needless to say, we're grateful for the deluge of orders. To handle it has taken a new office, new personnel, and new shipping systems; even then, the mass of orders -- a fifty times increase -- caused some delays. If your order didn't arrive quickly, thank you also for your patience. We believe you'll find JRT is worth the wait.

With the new capabilities, the goal of a one week order turn-around is now is sight.

Note 1: Five and a quarter inch disk versions

Requiring only 85K of diskette space for the compiler and 35K for the run-time system, JRT is currently the most compact Pascal available for CP/M systems. For program development in JRT Pascal on computers with five inch disk drives, we recommend this file arrangement:

On disk A:	On disk B:
- EXEC.COM	- JRTPAS2.COM
- your editor (ED, Wordstar, etc.)	- PASCAL.LIB
- the Pascal source program being developed	- PASCAL0.INT
	- PASCAL1.INT
	- PASCAL2.INT
	- PASCAL3.INT
	- PASCAL4.INT

IMPORTANT NOTE - The file PASCAL.LIB must always be present on the computer system when compiling or executing programs.

Note 2: Patch #1

Applicable version: 2.1
Error: multiplication of real numbers by 0.0 produces incorrect result
Patch procedure: Use CP/M program DDT to patch EXEC.COM - key in underlined code.

```
A>DDT EXEC.COM

DDT VERS 2.2
NEXT PC
5B00 0100
-S563C

563C ED EB

563D 53 .

-G0

A>SAVE 90 EXEC.COM
```

Note 3: Patch #2

Applicable version: 2.1
Error: Message 'Source file not found' when compiling under CP/M ver 1.4 or CDOS
Patch procedure: Use CP/M program DDT to patch JRTPAS2.COM - key in underlined code.

```
A>DDT JRTPAS2.COM

DDT VERS 2.2
NEXT PC
5500 0100
-A2B9

02B9  CALL 3F83

02BC  CALL 413D

02BF  .

-G0

A>SAVE 84 JRTPAS2.COM
```

Note 4: JRT Pascal version 2.2 update

Version 2.2 of JRT Pascal is now being shipped - 2.2 includes some internal enhancements and repairs all problems reported in earlier versions. If you want this update, it's yours for the cost of a diskette, postage and handling: $10.

The ONLY disk formats available are:

5 1/4" for Osborne, Apple CP/M, North Star, Superbrain, Heath hard sector, Heath soft sector, Xerox 820, Televideo

8" single-sided, single density standard

Please specify which of these formats you need.

Note 5: Coming - JRT Pascal version 3.0

In January we'll begin shipping JRT Pascal 3.0 - a major enhancement. New features include:

- builtin indexed file system
- facilities for screen and report formatting
- dynamic arrays
- improved compiler error recovery
- enhanced EXEC interrupt
- full support for file variables and GET/PUT
- expanded user manual

Of course the price of new 3.0 will still be $29.95.

Note 6: Copy and License Policy

We've had lots of questions about our policy on copying JRT Pascal. As our ads say, permission is granted to copy both disk and manual for friends - so long as it's not for resale.

Permission to make copies is also specifically granted to schools and to computer clubs for members.

If you develop application software for resale, you may distribute the run-time system (EXEC.COM and PASCAL.LIB) with your package - with no license or royalty fees.

Note 7: YOUR Pascal application programs

Naturally, more and more owners are developing more and more JRT Pascal written application packages for sale - we've heard from many of them. And - for developers - our copy and license policy is particularly attractive.

Now we're putting together a JRT Application Software Directory and would like to list the packages you have for sale. For free listing, just fill out the enclosed Application Program Description and return it to us with tangible evidence of your package such as brochure, manuals, diskette - but quickly, please: the first Directory is scheduled for February distribution.

Note 8: New address and phone number

The new phone number for orders only is (415) 566-5100.

The address for technical questions and problem reports:

JRT Systems
Technical Services
PO Box 22365
San Francisco, CA 94122

The address for new orders:

JRT Systems
550 Irving Street
San Francisco, CA 94122

Note 9: Feedback ... Please!

A dynamic product, new JRT Pascal versions are always being developed. The system's main evolutionary force is feedback from YOU - the user. We invite -- and encourage -- you to write us your ideas about how to make JRT Pascal even better.