SATURN SOFTWARE LIMITED

PRESENTS

>>>> SYM-PASCAL <<<<

RELEASE 2.0

BY

Ralph Deane

Distributed by:
SATURN SOFTWARE LIMITED
POST OFFICE BOX 397
NEW WESTMINSTER, BRITISH COLUMBIA
V3L 4Y7, CANADA

DISCLAIMER

SATURN SOFTWARE LIMITED and the author makes no warranties, either expressed or implied, with respect to this manual, or with respect to the software it describes, or its quality, performance, or suitability for any particular application. In no event will SATURN SOFTWARE LIMITED or the author be liable for any direct, indirect, incidental, or consequential damages resulting from any defects in the manual or software supplied.

While this software package is now operational of the author's 32K SYM-1/KTM-2 system, it should be expected that the purchaser may have to provide custom I/O drivers to match his/her particular terminal and printer configuration.

COPYRIGHT NOTICE

Address all communications to:

John W. Brown, President
SATURN SOFTWARE LIMITED
POST OFFICE BOX 397
NEW WESTMINSTER, BRITISH COLUMBIA
V3L 4Y7, CANADA

TABLE OF CONTENTS

TABLE OF CONTENTS

Introduction

What is Pascal ?

Pascal is not an acronym, unlike many languages. It was named after the mathematician Blaise Pascal (1623-1662) by the man who first specified the language, Niklaus Wirth. Pascal is an ideal block-structured programming language that inherently forces structured programming techniques and good programming practices. Writing an application in Pascal will reduce programming time by up to 50% for many problems. Program maintenence and the time spent adding program enhancements are also reduced because of Pascal's features.

What is SYM-Pascal ?

SYM-Pascal is a program which compiles and executes a subset of the Pascal language. This is done in two steps. First, the Pascal source program is compiled into an intermediate code similar to machine code, called P-code. The P-codes are then executed interpretively by the SYM-Pascal interpreter program. The interpreter is actually an idealized stack machine which is implemented in software instead of hardware, and whose native language is P-codes. Full compile-time (and execution time) error checking is done and, in spite of this, the compilation is extremely fast - approx. 50 to 100 lines per second. The program is also quite compact, with the editor, compiler and interpreter requiring only 8K of memory. SYM-Pascal can be run on machines with limited resources.

Equipment Required

SYM-Pascal requires the following minimum hardware in order to execute properly :

1. A SYM-1 computer with the RAE-1 ROM(s) installed.

2. At least 16K of RAM, addressed from $0000 to $3FFF.

3. A terminal such as a KTM-2/80 or equivalent.

4. One or two remote controlled cassette recorders, installed as per the RAE specifications.

5. (optional) A printer hooked to the SYM-1 20ma current loop.

I/O Customization

The following information should help you find the locations to change/patch for your preference or I/O setup.

$0200 Cold start entry point.
$0203 Warm entry point.
$020C New system Input Vector.
$020F New system Terminal Output Vector.
$0212 System bell vector, set to call SYM-beeper.
$0215 New system printer vector.

Note: All you I/O customization routines must use the above new vectors! SYM-Pascal hooks onto RAE by patching supermon and RAE vectors. This is done between $0218 and $0258 which can be dissassembled and modified with caution if found necessary.

$0258 Printer baud rate- currently set to $06 for 2400 baud.
$0259 Terminal baud rate- currently set to $01 for 4800 baud.
$025A KTMFLG- Currently set to $01 for KTM-2 type terminal.

Note: The only intelligent function your terminal must have is the ability to move back one character position on the screen when sent a backspace ($08).

The printer driver provided is the same as presented in the Super Terminal Patch for RAE from Softnews (01:03:06). The code is located at $025B through $02A0 and you may wish to disassemble and modify this routine rather than write your own from scratch.

You may hate the control codes provided with the line editor! The input line editor code is again essentially the same as in Softnews (01:03:06) and is located at $0310 through $049D. You may disassemble this, compare with what appeared in Softnews and change the control codes to suit your fancy!

System Startup

After loading the program from tape, the normal 'cold start' entry point is at $0200. Entry at this point makes no assumptions about text and/or P-codes in memory. The presence of the SYM-Pascal copyright message indicates that the program is running and waiting for input. Users with disk systems should boot SYM-Pascal after initializing their RAY disk patches (with RU $200). SYM-Pascal does not change the ENT LOD or DC vectors so you may use these commands to save and load Pascal source text and P-Codes to and from mass storage.

If, for some reason, you must exit the SYM-Pascal program it can be re-entered without affecting the text or P-codes. This is done by entering at location $0203, the 'warm start' address. A warm start should only be attempted after the system has been initalized by a cold start. Errors could occur otherwise. It should be noted that a warm start re-initalizes the tape control ports so that remote operation is still functional.

After both cold and warm starts, information about the state of

the system is output. The format of this data is :

Textstart-textend P-codestart
Textpointer

The textstart and textend addresses indicate the area set aside for the creation of the Pascal source program. The textpointer is the address of the actual end of the source text. The P-codestart address is the address at which the compiler will start depositing the P-code program.

Editor

One of the reasons that SYM-Pascal is so compact is that it 'sits on top' of the RAE system and uses its editor and file system in the preparation of the Pascal source program. The compiler accepts RAE compatible text as its input.

A slightly modified version of the Super Terminal Patch for RAE (by Jack Brown - Saturn Softnews 1:03) has been included in the SYM-Pascal editor as well as most RAE editing commands. Those RAE commands recognized by the SYM-Pascal editor are :

AUTO	(AU)	BREAK	(BR)	CLEAR	(CL)
COPY	(CO)	DELETE	(DE)	DUPLICATE	(DU)
EDIT	(ED)	FIND	(FI)	GET	(GE)
HARDCOPY	(HA)	MANUSCRIPT	(MA)	MOVE	(MO)
NUMBER	(NU)	OFF	(OF)	ON	(ON)
PRINT	(PR)	PUT	(PU)		
SET	(SE)	USER	(US)		

All other RAE commands will result in an !ED error. RAE's error messages are used. The commands EDIT (ED) and SET (SE) have been modified from the normal RAE operation. Their new actions are :

SET (SE)
-executing SE will output the state of the system in the format described in the Startup section.
-executing SE addr1 addr2 will reset the textstart address to addr1 and textend to addr2. An error will occur if both addresses are not present, although textstart will be changed. The error message is a warning that one address is missing.

EDIT (ED)
-executing ED line# will dump the contents of the line numbered line# to the input buffer where it can be edited using the line editor features.
-executing ED string will work in the normal RAE fashion.

Some new commands have been added to the SYM-Pascal editor which are not RAE compatible. These are :

PASCAL (PA)
-compile the Pascal program present in the text area. The P-codes are placed in memory starting at the P-codestart address. The P-codes are not executed.

EXEC (EX)
-compile as per PASCAL. Execute the P-code program. The run-time P-code stack starts at the end of the P-code program so memory must be avalible there.

Both PASCAL and EXEC print out the start and end address of the P-code program.

PSET (PS)
-change the P-codestart address. Used in the form :

PSET addr

where addr is the new P-codestart address.

GO (GO)
-execute the P-code program which starts at the P-codestart address.

SAVE (SA)
-save onto tape the P-code program which starts at the P-codestart address. Used in the form :

SAVE Fnumber

where number is the id number of the file. If Fnumber is omitted an id of 00 is used.

PLOAD (PL)
-load a P-code program from tape into memory, starting at the current P-codestart address. Used in the form :

PLOAD Fnumber

where number is the desired file id number. If Fnumber is omitted the next file found will be loaded.

It should be noted that P-code programs are address independant and can be executed from anywhere in memory. Therefore the load address does not have to be the same as the save address.

The line editor uses control codes to allow you to edit an input line. These codes are :

Control-A
-move cursor to start of current line and set buffer pointer to zero. This does not clear or cancel the current line of text.

Control-B
-insert a null ($00) into the text line and terminate the edit operation. This is used to insert a blank line into the text file. The compiler ignores all non-printable characters.

Control-C
-cancel or exit auto line mode. This equivalent to the standard // (ret) AU (ret) used by RAE.

Control-H
-backup the cursor one position on the screen and in the buffer. This has no effect on the buffer's contents but only serves to move the cursor back in a line for possible insertions or deletions. If the cursor is already at the start of the line, it will wrap around and appear at the end of that line.

Control-I
-tab forward one position in the buffer (and the screen) updating the current screen position if necessary. Tabbing past the end of the line will cause the cursor to wrap around to the start. This command is the opposite of Control-H.

Control-J (LineFeed)
-linefeed will put you in auto line mode with a step size of 1. If linefeed is typed with text in the buffer, the line is processed as normal before the next line number prompt is output.

Control-M (CarriageReturn)
-terminate the editing and send the line to SYM-Pascal for processing. The line is truncated at the current cursor position.

Control-P
-toggle the printer on and off. When hardcopy is set, using this command, the line count is set to 4 and the page count to 0.

Control-S
-escape to the SYM monitor. To return to Pascal type G (ret). Monitor I/O is sent to the printer if the hardcopy flag is set.

Control-T
-toggle the cassette motors on and off. No characters are echoed to the terminal but otherwise this is identical to RAE's Control-T.

Control-X
-cancel the current input line and clear the buffer. Text can now be re-entered into the line.

Control-Y
-exit to the monitor for one command. On completion you are returned to Pascal.

Control-Z
-move the cursor to the end of the line, updating the screen if necessary, and then erase to the end of the input line.

Escape (ESC)
-followed by any other character - skip through buffer and leave cursor at next location after first occurance of that character.

Delete (DEL)
-delete character to the left of the cursor position and close up that space in the buffer. The screen is not updated.

All other control characters are ignored. All non-control characters are inserted into the buffer at the cursor location. The screen is not updated but the characters are echoed to the terminal.

The line editor is designed to work with a terminal similar to the KTM-2, but can be adapted to a non-cursor oriented terminal. This is done by setting the value of KTMFLG ($025A) to $00. The only command which now does not work is Control-H. An input of this command is ignored. Delete (DEL) will also output a '\' to indicate that a character was deleted.

The SYM-Pascal line editor is essentially the same as the one included in SYM-FORTH and Extended SYM BASIC and should present no problems to users of those packages. Practice is needed to get the hang of using the proper command so don't despair.

Pascal Subset

The following material will explain the details of the Pascal subset supported by SYM-Pascal. We will begin by presenting a sample Pascal program and then explain each part in detail. This will be followed by a detailed explanation of each SYM-Pascal feature.

The program listed here is a simple line editor. It inputs characters from the terminal and builds a complete line of text in vector TEXT. Three editing commands are recognized by the editor. These are :

Control-H
-delete the last character entered into TEXT.

Control-X
-cancel the current input line, clear TEXT and start over again.

Carriage Return
-terminate the editing session. The contents of TEXT are then printed out.

All other non-printing characters are ignored. Printable data is inserted sequentially into TEXT, up to a maximum of 80 characters.

```
0010 (* A Line Editor written in SYM-Pascal *)
0020
0025 CONST CNTRLX=$18; CNTRLH=$08;
```



```
0030       CR=$0D; SPACE=$20 ;
0035
0040 VAR TEXT : ARRAY[79] OF INTEGER ;
0045     INDEX,CHARACTER,FLAG,TEXTPTR : INTEGER ;
0050
0055 PROC BACKSPACE ;
0060  BEGIN
0065   IF INDEX>0 THEN
0070    BEGIN
0075     INDEX:=INDEX-1 ;
0080     TEXT[INDEX]:=SPACE ;
0085     WRITE(CNTRLH@) ;
0090    END ;
0095  END ;
0100
0105  PROC CLEARTEXT ;
0110   BEGIN
0115    FOR TEXTPTR:=0 TO 79 DO
0120     TEXT[TEXTPTR]:=SPACE ;
0125   END ;
0130
0135 PROC CANCELINE ;
0140  BEGIN
0145   CLEARTEXT ;
0150   INDEX:=0 ;
0155   WRITE(\) ;
0160  END ;
0165
0170 PROC CARRIAGERETURN ;
0175  BEGIN
0180   FLAG:=1 ;
0185   INDEX:=INDEX-1 ;
0190   WRITE(\) ;
0195  END ;
0200
0205 PROC ENTERTEXT ;
0210  BEGIN
0215   IF CHARACTER>=SPACE THEN
0220    BEGIN
0225     TEXT[INDEX]:=CHARACTER ;
0230     WRITE(CHARACTER@) ;
0235     INDEX:=INDEX+1 ;
0240    END ;
0245  END ;
0250
0255 BEGIN (* MAIN *)
0260
0265  CLEARTEXT ;
0270  INDEX:=0 ;
0275  FLAG:=0 ;
0280
0285   REPEAT
0290
0295    READ(CHARACTER@) ;
0300    CASE CHARACTER OF
```

```
0305     CNTRLX : CANCELINE ;
0310     CNTRLH : BACKSPACE ;
0315     CR     : CARRIAGERETURN
0320     ELSE ENTERTEXT
0325    END ;
0330
0335   UNTIL (FLAG=1) OR (INDEX>79) ;
0340
0345   IF INDEX>79 THEN
0350    INDEX:=79 ;
0355
0360   FOR TEXTPTR:=0 TO INDEX DO
0365    WRITE(TEXT[TEXTPTR]@) ;
0370
0375 END . (* MAIN *)
0380
```

The explaination of each section is :

Line 10

This is a comment line giving the title of the program. A comment is any text enclosed by (* and *) and can be used anywhere.

Line 25 to 30

In this section we are defining constant values. The names CNTRLX, CNTRLH, CR and SPACE are all assigned specific values - CNTRLX equals (hex) 18, CNTRLH equals (hex) 08, CR equals (hex) 0D and SPACE equals (hex) 20. These names can now be used in place of the actual values throughout the program, aiding in readability. The definition of constants must be the first section (except for comments) in any Pascal program.

Line 40 to 45

This section is the declaration of variables. All variables must be declared before they can be used. In this declaration we are defining TEXT to be an array of 1 dimension (a vector) with 80 elements, those being numbered 0 to 79. We also define INDEX, CHARACTER, FLAG and TEXTPTR to be scalar integer variables. Variables and arrays have undefined values after they are declared. All variable declaration must follow the constant section.

After all constants and variables have been declared, we can start defining the subroutines to be used by our program. These subroutines are called procedures in Pascal and must be defined prior to their use.

Line 55 to 95

This is a procedure called BACKSPACE. What BACKSPACE is supposed to do is delete the last character entered into TEXT.

First, it checks to see if we are at the start of the line (ie. INDEX=0). If not it then executes a sequence of instructions. This sequence decrements INDEX by 1 so that it points to the last entered character, puts a space into that location in TEXT, and outputs a Control-H to the terminal. The procedure then terminates, returning control to the calling routine.

Line 105 to 125

This is a procedure called CLEARTEXT, which sets every element in TEXT to blanks. This is done using a FOR loop. TEXTPTR is initally set to 0 and is incremented by 1 on every pass through the loop. The loop ends when TEXTPTR is greater than 79. The body of the loop sets each element of TEXT, as indexed by TEXTPTR, to a blank.

Line 135 to 160

This procedure is called CANCELINE. It clears TEXT to blanks (using CLEARTEXT) and resets INDEX to 0. It then outputs a carriage return/line feed to the terminal.

Line 170 to 195

CARRIAGERETURN is the name of this procedure. It sets FLAG to 1, decrements INDEX by 1 and does a carriage return/line feed on the terminal.

Line 205 to 245

The procedure ENTERTEXT tests CHARACTER to see if it is a printable character (ie. ASCII value greater than or equal to SPACE). If this is true, it puts CHARACTER into the next location in TEXT, increments INDEX by 1, and outputs CHARACTER (as an ASCII character) to the terminal.

You will notice that the names of all the procedures are a good indicator of their function. SYM-Pascal allows names of procedures variables and constants to be any length, although only the first 8 characters are significant. This feature makes SYM-Pascal programs very readable even without comments.

Once all of the subroutines are defined we can define the main body of the program. The main portion of the program is enclosed inside the words BEGIN END . and does the following :

Line 265 to 275

This is the initalization section. TEXT is set to blanks and both INDEX and FLAG are zeroed.

Line 285

This is the start of our input loop. The loop construct used

is:

```
REPEAT <statements> UNTIL <condition-true>
```

In this loop we will input a character from the terminal and process it using our previously defined procedures. The loop will terminate if TEXT is full or FLAG equals 1.

Line 295

This inputs one ASCII character from the terminal and puts it in CHARACTER. The @ in the READ statement indicates character input. The input character is not echoed to the terminal.

Line 300 to 325

This CASE statement is used to process the input character. CHARACTER is tested by CASE and the following happens :

If CHARACTER=CNTRLX then execute the CANCELINE procedure.

If CHARACTER=CNTRLH then execute the BACKSPACE procedure.

If CHARACTER=CR then execute the CARRIAGERETURN procedure.

Otherwise execute the ENTERTEXT procedure.

Line 335

This tests to see if the loop should end. The test performed is if FLAG=1, indicating the input of a carriage return, or INDEX>79, indicating that TEXT is full. If either of these conditions is true the loop terminates. Otherwise execution continues at REPEAT.

Line 345 to 350

Once the loop terminates it is necessary to make sure that INDEX is not greater than 79. If this is true then INDEX is set to 79.

Line 360 to 365

The contents of TEXT is now output using a FOR loop. The loop will terminate when TEXTPTR is greater than the value of INDEX.

You will notice that this program is easy to read and understand even without comments. The structured programming techniques forced on you by Pascal ensure this. Spaghetti code is hard to write in SYM-Pascal due to the absence of a GOTO statement. A nicely structured program is also easy to enchance. For instance, we could add new commands to our line editor program simply by adding new procedures and more tests in the CASE statement. Nothing could be simpler.

Now that the walk through of a sample Pascal program has hopefully started your understanding of SYM-Pascal, we will look deeper at each aspect of the subset.

Character Set

The SYM-Pascal character set consists of the following :

```
letters  A-Z
numbers  0-9
special characters  @ + - * / = < > ( ) \
                    [ ] . , ; : ' " % $
                    and space.
```

Names

Names in SYM-Pascal consist of letters and/or numbers and may be any number of characters long. The first character must be a letter and the first 8 characters must be different than the first 8 characters of any other name. The following names are reserved by the Pascal language and cannot be user defined :

```
AND     ARRAY   BEGIN   CALL    CASE    CONST
DIV     DO      DOWNTO  ELSE    END     FOR
FUNC    IF      INTEGER MEM     MOD     NOT
OF      OFF     ON      OR      PRINT   PROC
READ    REPEAT  SHL     SHR     THEN    TO
UNTIL   VAR     WHILE   WRITE
```

Numbers

All numbers in SYM-Pascal are integer numbers (16 bit) with a range of -32768 to +32767. There are three forms of integer numbers - decimal, hexidecimal and character. Decimal numbers are a string of digits such as :

```
700
5192
3708
```

Hex numbers are a string of digits or letters A-F preceded by a $. Examples are :

```
$20
$6C9A
$00FE
```

Character numbers are a single character enclosed in quotes ('). The numerical value of these numbers is the ASCII value of the character. Example :

```
'A'    ($41)
'@'    ($21)
'\'    ($5C)
```

Comments

Any text enclosed between (* and *) is a comment. Comments can be placed anywhere in the program and are ignored by the compiler. Some examples are :

```
(* This is a comment *)
(* Start of Main Program *)
```

Integer Operations

The integer operations provided by SYM-Pascal are (in ascending order of precedence) :

```
  +  : addition
  -  : subtraction
  *  : multipication
 DIV : division
 MOD : modulo (remainder after division)
 SHL : bitwise shift left
 SHR : bitwise shift right
```

Examples :

-addition

```
5+1
NAME+TYPE
SCORE+1
```

-subtraction

```
4-2
NAME-1703
$9E0-SCORE
```

-multipication

```
74*41
TYPICAL*31
'B'*'@'
```

-division

```
1031 DIV 10
NAME DIV 200
SCORE DIV ':'
```

-modulo

```
17 MOD 5
SCORE MOD TYPE
```



```
TYPICAL MOD $19
```

-shift left

The value to the left of the operator is shifted left the number of bits specified by the value to the right.

```
10 SHL 2    (shift 10 left 2 bits)
SCORE SHL TYPE
TYPICAL SHL $0A
```

-shift right

The value to the left of the operator is shifted right the number of bits specified by the value to the right.

```
10 SHR 1    (shift 10 right 1 bit)
TOP SHR SCORE
TYPICAL SHR MAX
```

If the right side value is negative in either SHL or SHR then the other direction shift is used - ie:

```
10 SHR -2  is  10 SHL 2
50 SHL -9  is  50 SHR 9
```

Logical Operations

The logical operations provided are, in ascending order of precedence :

```
 OR  : bitwise logical OR
 AND : bitwise logical AND
 NOT : bitwise logical NOT
```

Examples :

-logical OR

```
255 OR $100
NAME OR SCORE
```

-logical AND

```
$CF AND $87
TYPE AND TOP
```

-logical NOT

```
NOT 0    (equals 1)
NOT 100  (equals 0)
```

Comparison

These operations (also called relational operators) result in a 0 if false and a 1 if true. The operators provided are :

```
=   : true if equal
<   : true if less than
>   : true if greater than
<>  : true if not equal
<=  : true if less than or equal to
>=  : true if greater than or equal to
```

Examples :

```
A=B
NAME<>SCORE
TYPE>=TOP
NAME<='Z'
```

Declaration

At the start of a program there is the declaration section, in which all objects local to the program are defined. These objects must be declared in a specific order, as follows :

1. constant definition
2. variable declaration
3. procedure and function declaration

We will look at each object separately and in detail.

1. Constant Definition

A constant definition introduces a name as a synonym for a numeric value. The use of constants generally makes a program more readable and acts as a convenient documentation aid. It also allows you to group system dependant values at the beginning of the program where they can be easily noted or changed.

The word CONST introduces the constant definition part, which uses the form :

```
CONST <name> = <value> ;
```

A series of constants can be defined by repeating the <name> = <value> section, separating each definition by a semicolon (;). The word CONST can appear only once in a declaration section. The definition can spread over more than one line. The value assigned can be either an unsigned number or a previously defined constant name.

Example :

```
CONST TYPE10=100 ;

CONST HEX10=$10 ;

CONST CHAR='A' ; CHAR9=CHAR ;
      BETA='B' ;
```

2. Variable Declaration

Every variable occuring in a program must first be declared in a variable declaration. This must precede any usage of that variable. A variable declaration associates a name and type with a new variable simply by listing the name followed by the type. SYM-Pascal has two variable types - scalar integer and 1-dimensional integer arrays.

The word VAR heads the variable declaration section and is in the form :

```
VAR <name> : <type>
```

A series of variables of the same type can be declared by :

```
VAR <name1>,<name2>, ... ,<nameN> : <type>
```

The <name> : <type> portion can be repeated more than once in a variable declaration as long as they are separated by semicolons (;). As with CONST the word VAR may appear only once in the declaration section. The declaration may spread over more than one line.

The <type> portion of the declaration can be either :

```
INTEGER     : for scalar integer or

ARRAY[<size>] OF INTEGER     : for integer arrays
```

In the array type <size> can be an unsigned number or a constant defined name. An array's elements start at 0, so a <size> of 10 will declare an array with 11 (0 to 10) elements.
Example :

```
VAR I,J,K : INTEGER ;

VAR VECTOR : ARRAY[75] OF INTEGER ;

VAR XYZ : INTEGER ;
    FILE : ARRAY[TYPE10] OF INTEGER ;
    KLM,JKL : INTEGER ;
```

The value of a variable is obtained simply by refering to it by

name. In the case of scalar integers this is all that is needed. Integer arrays need to be indexed as well. This is done by :

```
<array name>[<expression>]
```

where <expression> is any valid SYM-Pascal expression. No check is made to see if the size of the index is in bounds so care must be used.
Example :

```
I
XYZ
VECTOR[17]
FILE[J+K]
```

3. Procedure and Function Declaration

A. Defining a Procedure

It may happen that a particular set of actions has to appear several times in a program. We can avoid writing out the statements each time if we group them into a procedure. A procedure gives a name to a set of actions, which may then be called by referring to the name. A procedure may be considered as a program dedicated to some subtask of the overall problem.

The word PROC is used to identify the procedure declaration. In its simplest form it is like :

```
PROC <name> ;
```

The body of the procedure follows after this declaration. As an example, here is a procedure called DRAWLINE which outputs a line consisting of 10 '-' in a row :

```
PROC DRAWLINE
 CONST LENGTH=10 ;
 VAR I : INTEGER ;
 BEGIN
  FOR I:=1 TO LENGTH DO
   WRITE('-') ;
  WRITE(\)
 END ;
```

You'll notice that the body of the procedure is a complete program in itself, with a declaration section and all. It is important to note that any variables, constants, procedures or functions defined inside a procedure are 'local' to that procedure and are inaccessable from the outside. Anything defined or declared outside the procedure body is considered 'global' and can be accessed from the procedure body. If a local and global object have the same name, reference will always be to the local name (from inside the procedure body, of course).

The usefulness of a procedure is enchanced if its action can be varied from call to call. This is achieved by use of parameters. In SYM-Pascal these are variables used inside the procedure, which are given a different starting value with each call. These variables are local to the procedure. Parameters are specified in the declaration by :

```
PROC <name>(<parm1>,<parm2>, ... ) ;
```

where <parm1>, <parm2> etc. are valid scalar integer variable names. As an example, suppose DRAWLINE was to output a different length and character line on each call. It could be done as :

```
PROC DRAWLINE(LENGTH,CHAR) ;
 VAR I : INTEGER ;
 BEGIN
  FOR I:=1 TO LENGTH DO
   WRITE(CHAR@) ;
  WRITE(\)
 END ;
```

LENGTH and CHAR are automatically defined as scalar integer variables local to DRAWLINE. These are known as 'value parameters', since only an inital value is passed to the procedure. Values cannot be passed back to the calling routine except through global variables.

B. Defining a Function

A procedure is used to identify a set of actions. A function identifies an expression by associating a name (and parameters) with the calculation of a value. The name of the function, and its parameters, may be used wherever a variable or constant value might be used.

The word FUNC is used to introduce a function declaration. Everything else about a function is identical to a procedure except that the function name must be assigned a value somewhere in the function body. For example, a function MAX whose value is the larger of its two parameters :

```
FUNC MAX(X,Y) ;
 BEGIN
  IF X>Y THEN
        MAX:=X
  ELSE  MAX:=Y
 END ;
```

The function name must be assigned a value in the program body or the function value will be undefined. The value is considered to be scalar integer.

Program Body

The body of a program is a series of statements bracketed by BEGIN and END. The statements are separated by semicolons (;). SYM-Pascal has many different types of statements and we will examine each in detail.

A. The Assignment Statement

The most fundamental of all statements is the assignment statement. It specifies that a newly computed value be assigned to a variable. The form of an assignment is :

<variable> := <expression>

where := is the assignment operator, not to be confused with the relational operator = .

The value assigned to the variable is obtained by evaluating an expression. An expression consists of :

1. constant operands
   -these are either a number or a constant defined name.

2. variable operands
   -these are a variable name (and subscript if necessary).

3. operators
   -these are Integer, Logical and comparision operators.

4. function designators
   -these are a function name and parameter list.

5. memory load
   -this is like BASIC's PEEK command and is represented by :

   MEM[<expression>]

where <expression> evaluates to the address of the desired byte.

Brackets ('(' and ')') may be freely used in an expression. The conventional rules of left to right evaluation and operator precedence are observed in an expression.
Examples :

```
ROOT1 := SPACE+27
BYTE := MEM[ALPHA+10] DIV 4
DEGREE := DEGREE+10
FILE[10] := MAX(X,Y)+$110F
VECTOR[I] := VECTOR[J]*100
FLAG := (X>1) OR (Y>1)
```


Note that the assignment statement is very free-form. Spaces may be inserted as needed and the assignment may continue onto more than one line. The only restriction is that words can not be broken in the middle.

B. The Compound Statement

In SYM-Pascal, any place that a statement can be used, a compound statement may also be used. A compound statement is formed by the word BEGIN, a group of statements separated by ; and followed by the word END. Note that the main body of a program has the form of a compound statement.
Examples :

```
BEGIN
 SCORE := SCORE+SUM ;
 COUNT := COUNT+1
END

BEGIN
 X := (Y+Z) DIV 100 ;
  BEGIN
   T := (Q*75) MOD 81 ;
   F := N-18
  END
END
```

Note that the last statement pior to END does not need to be terminated by a semicolon.

C. The Procedure Call

A procedure is called by referring to the procedure name. Any parameters must be included, in the proper order and with the parameter list enclosed in brackets.
Examples :

```
BACKSPACE
DRAWLINE(15,'*')
```

In the call to DRAWLINE, LENGTH is set to 15 and CHAR to '*'. The parameters can also be an expression if so desired.

```
DRAWLINE(LINE*SPACE,CHARACT+16)
```

The proper number of parameters in the list must be provided or an error will occur.

D. The Machine Language Call

SYM-Pascal allows you to call machine language subroutines from inside a Pascal program. The form used for doing this is :

```
CALL(<expression>)
```

where <expression> evaluates to the address of the subroutine. All registers (A, X and Y) may be destroyed and the return to Pascal is made via a subroutine return, RTS.
Examples :

```
CALL($8000)
CALL(INPUT)
CALL(ROUT+OFFSET)
```

E. The Memory Store Statement

As well as fetching a byte from memory, we can also store a value into a memory byte. This is like BASIC's POKE command, and the form is :

```
MEM[<expression1>] := <expression2>
```

where <expression1> evaluates to the desired byte's address and <expression2> is the value to be stored there. Only the lower byte (range of 0 to 255) of the value is significant.
Examples :

```
MEM[$A000] := CODE+10
MEM[PARM] := MEM[PARM+2]
MEM[A+75] := VALUE
```

Since the machine language call cannot pass parameters, the MEM function can be used to load and store values to be used by the subroutines.

F. The Printer Control Statement

It is possible to turn the hardcopy flag on and off from a SYM-Pascal program. If the flag is set, all input and output to and from the Pascal program will be echoed to the printer. The hardcopy flag is set by :

```
PRINT-ON
```

and is reset by :

```
PRINT-OFF
```

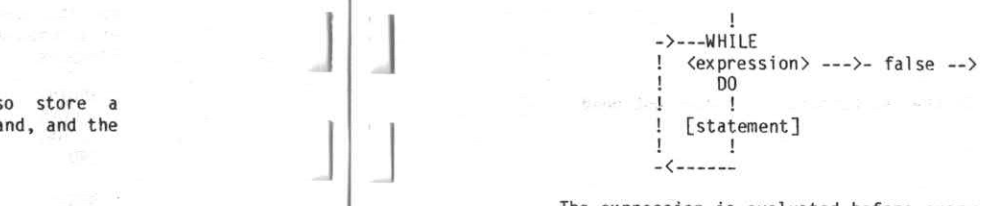
If the printer is turned on in a Pascal program but not off it will remain on upon return to the editor.

G. The WHILE Statement

The WHILE statement is used to repetitively execute a statement until a condition is false. The form of the WHILE statement is :

```
WHILE <expression> DO <statement>
```

The expression is considered false if it evaluates to 0. It is true otherwise. The <statement> portion is any valid SYM-Pascal statement including the compound statement. A flowchart of WHILE looks like :



The expression is evaluated before every iteration of the loop, so care must be taken to keep it as simple and fast as possible. The WHILE loop can iterate any number of times, including zero if the expression is false when first tested.
Examples ;

```
WHILE N>0 DO N := N-2

WHILE VECTOR[I]<>15 DO
  BEGIN
   MAX := VECTOR[I] ;
   I := I+1
  END
```

H. The REPEAT Statement

The REPEAT statement is used to repetitively execute a statement until a condition is true. The form of this statement is :

```
REPEAT <statements> UNTIL <expression>
```

The <expression> is the same as the WHILE statement. The <statements> portion is any number of valid Pascal statements, separated by semicolons. A flowchart of REPEAT looks like :

The expression is evaluated after every iteration of the loop and because of this the statements portion is always executed at least once. The loop can iterate any number of times, from one on up.
Examples :

```
REPEAT
 N := N+1
UNTIL N>75

REPEAT
 MAX := VECTOR[I] ;
 I := I+1
UNTIL VECTOR[I]=0
```

Notice that the last statement in the <statements> part does not need the trailing semicolon.

I. The FOR Statement

When the number of repetitions of a statement is known, the FOR statement can be used. This is very similar to the FOR-NEXT loop in BASIC, except that it is restricted to an increment of 1. The FOR statement indicates that a statement be repeatedly executed while a progression of values is assigned to the control variable. The value of the control variable increases by 1 throughout the progression. The form of the FOR loop is :

```
FOR <control variable> := <expression1> TO <expression2> DO
          <statement>
```

<expression1> evaluates to the inital value of the control variable, while <expression2> is the final value. These values are evaluated only one, at the start of the loop. The loop will terminate when the value of the control variable is greater than the final value. The control variable is any valid Pascal variable and must have been previously declared.
Examples :

```
FOR I:=0 TO 79 DO
    TEXT[I] := SPACE

FOR ALP := 100 TO INDEX DO
    BEGIN
     HAT := HAT DIV 3 ;
     ALP := ALP+1
    END
```

There is also a FOR statement in which the increment is -1. This has the form :

```
FOR <control variable> := <expression1> DOWNTO <expression2> DO
          <statement>
```


This is identical to the previous FOR loop except that the increment is -1 instead of 1. The loop terminates when the value of the control variable is less than the final value.
Examples :

```
FOR I := 75 DOWNTO 1 DO TEXT[I] := I+1

FOR POINT := START DOWNTO LENGTH DIV 4 DO
    OUTPUT(X,POINT)
```

J. The IF Statement

The IF statement specifies that a statement be executed only if a certain condition is true. If it is false, the either no statement or the statement following the word ELSE is executed. The two forms of the IF statement are :

1. IF <expression> THEN <statement>

2. IF <expression> THEN <statement1> ELSE <statement2>

In form 1 the statement part is executed only if the expression evaluates to a true value. In form 2, <statement1> is executed if the expression has a true value and <statement2> is executed if it has a false value.
Examples :

```
IF N>0 THEN A:=A+1

IF TEST THEN
 BEGIN
  A:=BET+10 ;
  TEST:=0
 END

IF JACK>=0 THEN
          SCORE := SCORE+1
 ELSE     SCORE := SCORE-1
```

K. The CASE Statement

The CASE statement selects a single statement for execution from its component statements. The CASE statement consists of an expression (the selector) and a list of statements, each being labelled by a constant value(s). CASE selects for execution that statement whose label is equal to the selector value. If no such label is listed then either no statement or the statement following the word ELSE is executed. The two forms of CASE are :

```
1. CASE <expression> OF
        <label list> : <statement> ;
                 ......
        <label list> : <statement>
   END
```

```
2.  CASE <expression> OF
        <label list> : <statement> ;
                ......
        <label list> : <statement>
        ELSE <statement>
    END
```

The <label list> consists of one or more constant values separated by commas (,). If the selector value equals any one of the values in the label list then that statement is executed. In form 2, if no label match is made then the statement following ELSE is executed.
Examples :

```
CASE NUMBER OF
     10 : A:=A+1 ;
     11 : A:=A+2
END

CASE (ALPHA*BETA) OF
     CONS : CONTROL(X,Y) ;
     17,VAL : TEST(Z) ;
     6,$10 : A:=BETA
     ELSE TEST(A)
END
```

L. The WRITE Statement

The WRITE statement outputs values evaluated from expressions. It also has some formatting of the output available. The basic form of the WRITE statement is :

```
WRITE(<value1>, ...  ,<valueN>)
```

where <value> can be a format operator, an expression or a quoted string.

The stand-alone format operators are \ and " . Their actions are :

\ : output a carriage return/line feed to the terminal. More than one \ may be grouped together to be treated as a single operator.

" : this is used to set the output field width to the value of the following constant.

Examples :

```
WRITE(\)            -output one CR

WRITE(\\\)          -output three CR

WRITE("10)          -set field width to 10
```



```
WRITE("5)           -set field width to 5
```

The field width can be set more than one time in a WRITE statement and will remain the same until :

a. it is reset by a " operation
or b. the WRITE statement ends.

The default value of the field width is 1. This is the minimum number of characters output for every value. If the value requires less characters, it is preceded by a suitable number of spaces. If it requires more characters, all the characters will be output. The entire value is always output.

When the result of an expression is output, format operators are used to indicate the type of output. These operators are :

@ : treat the value as one ASCII character.

% : output value as a hex number.

If no format operator is specified, the value is output as a decimal number. The @ or % is appended to the end of the expression.
Examples :

```
WRITE($41@)             -output the ASCII character A.

WRITE(SCORE)            -output as decimal number.

WRITE(VALUE%)           -output as hex number.
```

It should be noted that the character output (@) does not use field width. Only one character is output and will require only one space.

Messages are output as quoted strings. This is a string of characters (maximum 80) enclosed by ' . No field width is used.
Examples :

```
WRITE('Message')

WRITE('PLEASE HELP')
```

By combining the different types and formats in the WRITE statement, a very versatile output feature is available.
Examples :

```
WRITE('The value of A+B is ',"10,A+B)

WRITE(\\,VAAL%,'is equal to',VAL,\)
```

M. The READ Statement

The READ statement is used to input values from the terminal and place them into Pascal variables. Formatting of the input is also

available. The general form of the READ statement is :

```
READ(<value1>, ... ,<valueN>)
```

where <value> can be a variable name, format operator or quoted string.

The only stand-alone format operator is the \ operator, which outputs a carriage return/line feed to the terminal. This is the same as the \ operator in the WRITE statement. More than one \ may be grouped together. The quoted string is output as a message and is identical to a WRITE statement message.

When <value> is a variable name, format operators are used to indicate the type of input desired. The operators are :

@ : input one keystroke from the terminal as an ASCII character. The character is not echoed to the terminal.

% : input as a hex number.

No format operator will cause the input to be a decimal number. With both hex and decimal number input, any error will cause the system to backspace over the input field and restart the input. The number may be signed (+ or -) and is terminated by typing a carriage return. A space is output at the end of each input field to delimit multiple inputs.
Examples :

```
READ(ALPHA,BETA%)

READ('TEST = ',TEST@)

READ(\\,VECTOR[I],'is the value',\)
```

Summary

This ends the discussion of statements used inside a program body. If the program body is used in a procedure or function it must be followed by a ; to indicate the end. This would look like :

```
BEGIN
 <program body>
END ;
```

If the program body is the main body it must be followed by a period (.). This will halt the compiler properly. It would look like :

```
BEGIN
 <program body>
END .
```


Recursion

All SYM-Pascal procedures and functions are fully recursive. This means that a procedure or function can call itself from within the program body. A new set of local variables are generated with each call so nothing will be disturbed in the calling routine. The QUICKSORT program in the Sample Program section is an example of recursion. Care must be taken when using recursion to insure that the recursion will eventually end and that the P-stack does not overflow memory. Recursion is usually slow and greedy for memory.

Compiling from Tape

It is possible to break a large Pascal program into pieces and then have SYM-Pascal compile it from the cassette tape machine. This is accomplished via the .T pseudo-command. When the compiler encounters .T in a Pascal source program it will load the next source file from tape, into the Pascal text area, and then continue compiling at the start of that file. A forced load of the next file is used to get the source file into memory - ie. an id of 00 is used. The text area must be big enough to hold the largest program segment or an error will occur.

By breaking a program down into smaller pieces, a large Pascal program can be compiled on a machine with a limited amount of memory. For really large programs the following steps are necessary to compile and execute :

A. compile from tape using PA and .T .

B. save the P-code program using SA .

C. change the P-code start address to the start of the largest continuous block of usable memory by use of PS .

D. load the P-code program using PL .

E. execute the program using GO .

This sequence allows you to compile and execute programs which are normally too large for the system.

Error Messages

SYM-Pascal has extensive error checking in the editor, compiler and interpreter. In addition to the RAE editor error messages there are four new error messages which use the RAE format. These are :

Error Number	Cause
!40	out of memory
!50	attempted divide by 0
!EF	missing address in SET or PSET
!F0	P-code program not in P-code area

The compiler lists the errors in a different format. Included is the error number, the line in which it occured and the word nearest to the error location. The compile-time errors are :

Error Number	Cause
02	constant expected
03	= expected
04	name expected
05	; or : missing
09	. expected
10	UNTIL expected
11	undeclared name
12	illegal name
13	:= expected
16	THEN expected
17	; or END expected
18	DO expected
19	incorrect symbol
20	comparison operator expected
21	use of procedure name in expression
22	) expected
23	illegal factor
25	BEGIN expected
26	OF expected
28	TO or DOWNTO expected
30	missing parameters in call
31	(expected
33	[expected
34	] expected
35	parameters mismatched
36	data type unknown
37	, expected
38	ON or OFF expected
39	*) missing
41	number expected
42	field width out of range (1 to 255)
43	null string not allowed

All compile time errors are fatal and cause a reentry into the editor. A program with errors can not be executed, even via GO. This is a added safty feature of SYM-Pascal.

The P-Machine

It is not necessary to understand the inner workings of the P-machine in order to use SYM-Pascal. The information provided here is for general interest only, as the running of the P-machine is transparent to the user.

The interpreter portion of SYM-Pascal emulates, in software, an idealized stack machine whose native (ie. machine) language is P-codes. This P-machine is a stack oriented processor consisting of four registers and two memory storage areas. Memory is separated into program storage and data storage areas. The program storage area contains the P-code program and remains unchanged during program execution (ie. no self-modifying code). The data storage area contains the values of the variables and is also used for temporary storage during arithmetic and logical operations. Though the variables can be fetched and stored in a random fashion, the data storage area operates as a stack with respect to arithmetic/logical operations and run-time variable storage allocation. For this reason the data storage area is known as the P-stack. The P-stack is located immediately following the program storage area in memory and grows upwards. The four registers in the P-machine are :

a. the program counter, which points to the next executable P-code in the program storage area.

b. the instruction counter, which contains the current P-code.

c. the stack pointer, which points to the top of the P-stack.

d. the base register, which points to the start of the local variable storage area.

All storage allocation is dynamic (ie. as it is needed), so variable addresses cannot be assigned at compile time. Instead they are generated as offsets from the base register. This makes P-code programs address independent and means that they can be executed from anywhere in memory.

The P-machine has only 11 basic instuctions, each of which is four bytes long. Variations on these intructions give a total of 43 different operations. In the following description of the P-codes, these letters stand for :

V : static level difference
D : displacement from base register
N : 16 bit constant value
A : 16 bit address, with 0 being the start of the P-code program.
B : 8 bit data byte, with range of 0 to 255

The top stack element is denoted by (sp), while the second element is denoted by (sp-1).

The P-machine instruction set is :

Mnemonic	Description
LIT 0 N	copy N onto stack
OPR 0 0	procedure return
OPR 0 1	(sp) := -(sp)
OPR 0 2	(sp) := (sp)+(sp-1)
OPR 0 3	(sp) := (sp-1)-(sp)
OPR 0 4	(sp) := (sp)*(sp-1)
OPR 0 5	(sp) := (sp-1)/(sp)
OPR 0 6	(sp) := low order bit of (sp)
OPR 0 7	(sp) := (sp-1) modulo (sp)
OPR 0 8	test (sp-1)=(sp)
OPR 0 9	test (sp-1)<>(sp)
OPR 0 10	test (sp-1)<(sp)
OPR 0 11	test (sp-1)>=(sp)
OPR 0 12	test (sp-1)>(sp)
OPR 0 13	test (sp-1)<=(sp)
OPR 0 14	(sp) := (sp)OR(sp-1)
OPR 0 15	(sp) := (sp)AND(sp-1)
OPR 0 16	(sp) := NOT(sp)
OPR 0 17	shift (sp) left 1 bit
OPR 0 18	shift (sp) right 1 bit
OPR 0 19	(sp) := (sp)+1
OPR 0 20	(sp) := (sp)-1
OPR 0 21	copy (sp) onto stack
LOD V D	load a value
LOD 255 0	load a byte from absolute address (sp)
LODX V D	load a value with index address (sp)
STO V D	store a value
STO 255 0	store a byte at absolute address (sp-1)
STOX V D	store a value with index address (sp)
CAL V A	procedure call
CAL 255 0	machine language call to address (sp)
INT 0 N	increment stack pointer by N
JMP 0 A	jump to P-code address A
JPC 0 A	jump to P-address A if low order bit (sp)=0
JPC 1 A	jump to P-address A if low order bit (sp)=1
CSP 0 0	input one character, no echo
CSP B B1 1	output B by B1 times
CSP 0 2	input a decimal number
CSP 0 B 3	output a decimal number with field width B
CSP 0 4	input a hex number
CSP 0 B 5	output a hex number with field width B
CSP 0 6	output (sp) as an ASCII character
CSP B 8	output B characters from the following string

These 43 instructions are all that is needed to execute any program written in SYM-pascal. The small instruction set keeps the interpreter from being too slow, yet still does the job.

Note: the interpreter checks for a break key down everytime it executes another P-code instruction. If the break key was pushed, the program halts and waits for another keystroke. A Control-Z will

halt execution and return you to the editor. A Control-Q will resume execution. Everything else is ignored. The break key does not work while waiting for character or numeric input.

P-Code Decompiler

Presented here is a Pascal program which will decompile a p-code program. With it you can look at the codes produced after compiling your favorite program.

The first step in decompiling is to compile your program into memory. Then change the P-code start address to another area, where you will compile and execute the decompiler program. Watch out that the two P-code areas do not overlap. When the decompiler program asks for an address, enter the P-code start address of your program. The decompiled listing is output in 15 line blocks, with a pause for keyboard input in between. Any input but a Control-X will output the next block. A Control-X will cancel the decompiler.

The Pascal P-code Decompiler program listing is :

```
(* A P-code disassembler written in SYM-Pascal *)

CONST STOPCODE=$1F ;
VAR STARTADDR, LINENUM, NUMBER,LINECNT : INTEGER ;

PROC TEXT ; (* used by CSP b 8 *)
 VAR I :INTEGER ;
 BEGIN
  WRITE('  ') ;
  WRITE("3,MEM[STARTADDR+1],' ') ; (* output b *)
  WRITE("4,MEM[STARTADDR+2],\) ; (* output 8 *)
  WRITE('            ',$27@) ;
  FOR I:=1 TO MEM[STARTADDR+1] DO
   WRITE(MEM[STARTADDR+3+I]@) ; (* output following string *)
  WRITE($27@) ;
  LINENUM := LINENUM+MEM[STARTADDR+3] ;
  LINECNT := LINECNT+1 ;
  STARTADDR := (MEM[STARTADDR+3]*4)+STARTADDR ;
 END ;

PROC OPCODE ; (* used by all P-codes except CSP *)
 BEGIN
  IF (NUMBER=$12) OR (NUMBER=$13) THEN WRITE('X')
                                  ELSE WRITE(' ') ;
  WRITE(' ') ;
  WRITE("3,MEM[STARTADDR+1],' ') ;
  NUMBER := (MEM[STARTADDR+3]*256) + (MEM[STARTADDR+2] AND 255) ;
  WRITE("4,NUMBER)
 END ;
```

```
PROC CSPCODE ; (* used by CSP *)
 BEGIN
  CASE MEM[STARTADDR+2] OF
       0,2,4,6 : OPCODE ;
               8 : TEXT
       ELSE BEGIN
             WRITE('  ') ;
             IF MEM[STARTADDR+1]=255 THEN WRITE("3,$0D)
                                     ELSE WRITE("3,MEM[STARTADDR+1]) ;
             WRITE(' ') ;
             WRITE("4,MEM[STARTADDR+3],' ') ;
             WRITE("4,MEM[STARTADDR+2])
            END
  END ;
 END ;

BEGIN  (* Main Body *)
 READ(\,'Start address of P-codes (in hex) =$',STARTADDR%,\) ;
 LINECNT := 0 ;
 LINENUM := 0 ;
 NUMBER := 0 ;
 WHILE (NUMBER=0) AND (MEM[STARTADDR]<>STOPCODE) DO
  BEGIN
   WRITE("9,LINENUM) ; (* output P-code address *)
   WRITE('   ') ;
   LINECNT := LINECNT+1 ;
   LINENUM := LINENUM+1 ;
   NUMBER := MEM[STARTADDR] ; (* get P-code *)
   CASE NUMBER OF
        0 : BEGIN
             WRITE('LIT') ;
             OPCODE
            END ;
        1 : BEGIN
             WRITE('OPR') ;
             OPCODE
            END ;
    2,$12 : BEGIN
             WRITE('LOD') ;
             OPCODE
            END ;
    3,$13 : BEGIN
             WRITE('STO') ;
             OPCODE
            END ;
        4 : BEGIN
             WRITE('CAL') ;
             OPCODE
            END ;
        5 : BEGIN
             WRITE('INT') ;
             OPCODE
            END ;
```



```
        6 : BEGIN
             WRITE('JMP') ;
             OPCODE
            END ;
        7 : BEGIN
             WRITE('JPC') ;
             OPCODE
            END ;
        8 : BEGIN
             WRITE('CSP') ;
             CSPCODE
            END
   END ; (* end of CASE *)
   WRITE(\) ;
   NUMBER := 0 ;
   IF LINECNT MOD 15 =0 THEN  (* test if 15 lines output *)
    BEGIN
     READ(NUMBER@) ; (* input keystroke *)
     IF NUMBER=$18 THEN NUMBER:=1  (* Control-X cancels *)
                   ELSE NUMBER:=0 ;
    END ;
   STARTADDR := STARTADDR+4 ; (* get address of next P-code *)
  END ; (* end of WHILE *)
 WRITE('            EOF',\\) ;
END .  (* end of Main Body *)
```

Sample Programs

This section contains a group of programs written in SYM-Pascal. They are intended to illustrate various Pascal constructs and to show typical Pascal programming techniques.

MAX4

The program MAX4 asks you to input four numbers, separated by carriage returns. It will then print out the largest number of the four.

```
(* Find the maximum of 4 numbers *)

VAR NUM1, NUM2, NUM3, NUM4 : INTEGER ;

FUNC MAX(X1,X2,X3,X4) ; (* find max of 4 numbers *)
 VAR TEMP1, TEMP2 : INTEGER ;

 FUNC MAX2(Y1,Y2) ; (* find maximum of Y1 and Y2 *)
  BEGIN
   IF Y1>Y2 THEN MAX2 := Y1
            ELSE MAX2 := Y2
  END ;

 BEGIN (* start of MAX *)
  TEMP1 := MAX2(X1,X2) ; (* maximum of first two numbers *)
  TEMP2 := MAX2(X3,X4) ; (* maximum of second two numbers *)
```

```
  MAX := MAX2(TEMP1,TEMP2) (* maximum of two largest numbers *)
 END ; (* end of MAX *)

BEGIN (* Main *)
 REPEAT
  READ(\\,'The four numbers? ',NUM1,NUM2,NUM3,NUM4) ;
  WRITE(\,'The largest is ',"6,MAX(NUM1,NUM2,NUM3,NUM4),\)
 UNTIL NUM1<0 (* repeat until first number input is negative *)
END . (* Main *)
```

ADD

The program ADD asks for repeated input of numbers, each terminated by a carriage return. It will keep a running total of all the numbers input, and will print out this total when the input number is 0.

```
(* Add integers until a 0 is encountered *)

VAR SUM, NUMBER : INTEGER ;
BEGIN (* Main *)
 SUM := 0 ; (* set total to 0 *)
 REPEAT
  READ(\,'Number? ',NUMBER) ; (* input number *)
  SUM := SUM + NUMBER ; (* compute sum *)
 UNTIL NUMBER=0 ; (* loop until a 0 is input *)
 WRITE(\,'The total is ',"6,SUM) (* output total *)
END . (* Main *)
```

POWER

The program POWER will ask you for the input of two numbers, A and B. It will then find and print the largest number N such that B**N is less than A.

```
(* Find largest N such that B**N < A *)

VAR A, B, PRODUCT, POWER : INTEGER ;
BEGIN
 READ(\,'Value of A? ',A) ; (* input A *)
 READ(\,'Value of B? ',B) ; (* input B *)
 POWER := 0 ; (* N equals 0 *)
 PRODUCT := B ; (* set to B**0 *)
 WHILE PRODUCT <= A DO (* continue until B**N > A *)
  BEGIN
   POWER := POWER+1 ; (* increment N *)
   PRODUCT := PRODUCT * B (* compute B**N *)
  END ;
 WRITE(\,"2,'The largest power of ',B,' < ',A,' is',P R)
END . (* Main *)
```


AVERAGE

This program finds the average of a group of values. AVERAGE first asks for the number of values to be input. It then asks for each value, with the prompt 'Number?'. Terminate each number with a carriage return. After the correct number of values is input, the average is printed.

```
(* Form average of N numbers *)

VAR I, N, AVERAGE : INTEGER ;
    SUM, NUMBER   : INTEGER ;

BEGIN (* Main *)
 READ(\,'Number of values? ',N) ; (* input number of values *)
 SUM := 0 ;
 FOR I:= 1 TO N DO (* input N values *)
  BEGIN
   READ(\,'Value? ',NUMBER) ;
   SUM := SUM + NUMBER (* compute sum of values *)
  END ;
 AVERAGE := SUM DIV N ; (* compute average *)
 WRITE(\,'Average is ',"6,AVERAGE)
END . (* Main *)
```

GCF

The program GCF finds the greatest common factor of two numbers, A and B. GCF asks for the input of the A and the B values, and then prints out the greatest common factor.

```
(* Find greatest common factor of A and B *)

VAR A, B : INTEGER ;

BEGIN (* Main *)
 READ(\,'Value of A? ',A) ; (* input A *)
 READ(\,'value of B? ',B) ; (* input B *)
 REPEAT
  WHILE A > B DO
   A := A - B ; (* do until A <= B *)
  WHILE B > A DO
   B := B - A ; (* do until B <= A *)
 UNTIL A = B ; (* continue until A equals B *)
 WRITE(\,'The greatest common factor is ',"6,A)
END . (* Main *)
```

TOTALS

This program forms totals of all positive numbers, all negative numbers and counts zeros from numbers input to it. First, TOTALS

asks for the number of values. It then inputs the values. When all values have been input, it will print out the totals.

```
(* Form a count of positive and negative numbers and
    count the number of zeros *)

VAR I, N, NUMBER, COUNT : INTEGER ;
    POSSUM, NEGSUM      : INTEGER ;

BEGIN (* Main *)
 READ(\,'Number of values? ',N) ;
 POSSUM := 0 ; (* zero both sums *)
 NEGSUM := 0 ;
 COUNT := 0 ;
 FOR I:= 1 TO N DO (* loop for N values *)
  BEGIN
   READ(\,'Value? ',NUMBER) ;
   IF NUMBER = 0 THEN (* test for 0 *)
      COUNT := COUNT+1 (* increment 0 count *)
    ELSE (* test for positive or negative *)
     IF NUMBER > 0 THEN POSSUM := POSSUM + NUMBER
                   ELSE NEGSUM := NEGSUM + NUMBER
  END ;
 WRITE(\,'Total of positive numbers is ',POSSUM) ;
 WRITE(\,'Total of negative numbers is ',NEGSUM) ;
 WRITE(\,'Total number of zeros is ',COUNT)
END . (* Main *)
```

DAY

The program DAY asks for the number of a day of the week. It then outputs the name of that day. Illegal numbers are error trapped.

```
(* Read number, and print out corresponding day of the week *)

VAR DAYNO : INTEGER ;

BEGIN
 READ(\,'Input the number of day of week: ',DAYNO,\) ;
 CASE DAYNO OF (* test day number *)
   1 : WRITE('SUNDAY') ;
   2 : WRITE('MONDAY') ;
   3 : WRITE('TUESDAY') ;
   4 : WRITE('WEDNESDAY') ;
   5 : WRITE('THURSDAY') ;
   6 : WRITE('FRIDAY') ;
   7 : WRITE('SATURDAY')
   ELSE WRITE('Not a day of week number!')
 END ;
END . (* Main *)
```


CALCULATOR

This program makes Pascal act like a four function calculator. When input is asked for, you should input a number, terminated by a carriage return. Then input an operator (+ - * /) with no terminator. Continue entering number and operator until you want the answer. Then input = as the operator. The answer is printed out.

```
(* Hand calculator simulator *)

VAR OPERATOR, ANSWER, NUMBER : INTEGER ;

BEGIN (* Main *)
 ANSWER := 0 ; (* initalize values *)
 OPERATOR := '+' ;
 WRITE(\,'Input? ') ;
 REPEAT
  READ(NUMBER) ; (* input number *)
  CASE OPERATOR OF
    '+' : ANSWER := ANSWER + NUMBER ;
    '-' : ANSWER := ANSWER - NUMBER ;
    '*' : ANSWER := ANSWER * NUMBER ;
    '/' : ANSWER := ANSWER DIV NUMBER
  END ;
  READ(OPERATOR@) ; (* input operator *)
  WRITE(OPERATOR@,' ') ;
 UNTIL OPERATOR = '=' ;
 WRITE(\,'Answer is ',ANSWER) ;
END . (* Main *)
```

C2F

This program will print out a range of values in degrees centigrade along with the value in degrees fahrenheit. The range is 0 to 100 in steps of 5.

```
(* Convert degrees Centigrade to degrees Fahrenheit *)

CONST OFFSET=32 ; (* conversion offset *)
VAR CENTEMP, FAHRTEMP : INTEGER ;

BEGIN
 WRITE(\,'Centigrade to Fahrenheit',\) ;
 FOR CENTEMP := 0 TO 100 DO (* range of 0 to 100 degrees *)
  BEGIN
   FAHRTEMP := (CENTEMP * 9)  DIV 5 + OFFSET ;
   WRITE(\,"7,CENTEMP,"10,FAHRTEMP) ;
   CENTEMP := CENTEMP+4 ; (* make CENTEMP increment by 5 *)
  END
END .
```

HISTGRM

This program converts numeric data into a histogram. First, it asks for the number of values, which must be 15 or less. It then inputs the values. When finished, the histogram is output.

```
(* Histogram print routine *)

VAR NUMBER, N, I : INTEGER ;
    DATA : ARRAY[15] OF INTEGER ;

PROC DRAWLINE(LENGTH) ;
 VAR I : INTEGER ;
 BEGIN
  FOR I:= 1 TO LENGTH DO
   WRITE('*') ;
  WRITE(\)
 END ;

BEGIN (* Main *)
 REPEAT (* size must be less than 16 *)
  READ(\,'Number of values? ',N)
 UNTIL N<16 ;
 FOR I:= 1 TO N DO
  READ(\,'Value? ',DATA[I]) ; (* input data *)
 WRITE(\\) ;
 FOR I:= 1 TO N DO
  BEGIN
   NUMBER := DATA[I] ; (* get next value *)
   IF NUMBER > 0 THEN (* ignore data if not > than 0 *)
    IF NUMBER >79 THEN DRAWLINE(79) (* max size of 80 *)
                  ELSE DRAWLINE(NUMBER) ;
  END ;
END .
```

QUICKSORT

This program does a recursive QuickSort on a list of values. It first asks for the input of the values in the list (8 of them) and sorts them. The sorted list is then output.

```
(* The QuickSort sorting routine *)

CONST N=9 ; (* number of elements in list *)
VAR LIST : ARRAY[N] OF INTEGER ; (* storage area of list *)
    K : INTEGER ; (* temporary variable *)

PROC SORT(LOWER,UPPER) ;
 VAR I, J, TEMP, VALUE : INTEGER ;
 BEGIN
  J := UPPER ; (* set upper index value *)
  I := LOWER ; (* set lower index value *)
  VALUE := LIST[LOWER] ; (* inital comarision value *)
  IF LOWER < UPPER THEN (* make sure there is a section to test *)
```



```
  BEGIN
   WHILE I < J DO (* until lower index = upper index *)
    BEGIN
     WHILE LIST[J] > VALUE DO
      J := J-1 ; (* find smaller value *)
     WHILE LIST[I] <= VALUE DO
      I := I+1 ; (* find larger value *)
     IF I <= J THEN
      BEGIN
       TEMP := LIST[J] ; (* swap values *)
       LIST[J] := LIST[I] ;
       LIST[I] := TEMP
      END
    END ;
   LIST[LOWER] := LIST[J] ; (* swap values *)
   LIST[J] := VALUE
  END ;
  IF J > LOWER THEN
   SORT(LOWER,J-1) ; (* sort lower section *)
  IF J < UPPER THEN
   SORT(J+1,UPPER) ; (* sort upper section *)
 END ;

BEGIN (* Main *)
 WRITE(\,'Input',"2,N-1,' numbers to be sorted',\) ;
 FOR K := 0 TO (N-1) DO (* input N-1 numbers *)
  BEGIN
   WRITE(\,'Number',"2,K,'? ') ;
   READ(LIST[K])
  END ;
 WRITE(\) ;
 LIST[N] := 32767 ; (* set last element to largest number *)
 SORT(0,N-1) ; (* sort first N-1 elements *)
 WRITE(\\,'The sorted list is :',\) ;
 FOR K := 0 TO N-1 DO
  WRITE(\,"6,LIST[K]) ; (* output sorted list *)
END .
```