FLEX™ Pascal User's Manual

™ FLEX is a trademark of Technical Systems Consultants, Inc.

MANUAL REVISION HISTORY

Revision	Date	Change
A	2/81	Original Release, Pascal System Version 1.0
B	6/81	Effective: 6809 Pascal System Version 2.0 Change page numbering. p. 4 Add description of PASCDEF.TXT file. p. 5 Add mention of temporary files and ASN. p. 7 Add mention of uppercase in PREFIX. p. 10 Add varient record exceptions and EOF/EOLN syntax. p. 11 Add syntax for comments.

We would like to thank the Springer-Verlag Publishers for granting us permission to cite examples from their book, Pascal User Manual and Report, by Kathleen Jensen and Niklaus Wirth.

COPYRIGHT INFORMATION

DISCLAIMER

The supplied software is intended for use only as described in this manual. Use of undocumented features or parameters may cause unpredictable results for which Technical Systems Consultants, Inc. cannot assume responsibility. Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Technical Systems Consultants, Inc. will not assume responsibility for any damages incurred or generated by such material. Technical Systems Consultants, Inc. reserves the right to make changes in such material at any time without notice.

technical systems consultants, inc.

111 Providence Road Chapel Hill North Carolina 27514

IMPORTANT NOTE for 5" PASCAL

As you should already know, there are two characteristics of Technical Systems Consultants Inc's 6809 Native-Code Pascal Compiler that are special to the minifloppy version. The first is that it will not all fit on one single sided, single density diskette, requiring the user to switch disks to perform a compilation. Second is that the large amount of disk I/O performed by our compiler is more apparent on the slower minifloppies in the form of longer compilation times. If the user has a minifloppy disk systems which supports double sided or double density or double track density diskettes, these points cause no real problem as the user can immediately copy the files from the two supplied diskettes to one of his higher density system diskettes. If, however, the user has only a single sided, single density disk system, he should read the following for details of the special procedures required by the minifloppy diskettes.

A) Performing a Pascal compilation

The several modules required to perform a compilation take up the entire space of a single sided, single density diskette. This implies that the compiler cannot be copied to your FLEX "system" disk (the disk with all your command files). The run-time package (PRUN.CMD), which is not very large, can and should be copied onto your system disk for later execution of compiled Pascal programs. Assuming you have assigned drive #0 as your system drive and drive #1 as your working drive in a two drive system, you must take the following steps to perform a compilation:

1) Prepare your Pascal source file with an editor and place the diskette containing that file in drive #1. Be sure there is room left on the disk for temporary Pascal files.
2) Remove your system disk from drive #0 and insert the supplied diskette containing the Pascal compiler modules.
3) Execute the Pascal compiler.
4) Remove the compiler diskette from drive #0 and reinsert your system diskette. You may now edit the source file if there were errors or if no errors, execute the compiled program with the PRUN command.

B) Backing up the Pascal compiler diskette

Since the Pascal diskette containing the compiler modules is completely full, you may have difficulty in backing up the disk unless you have a "backup" or "mirror" utility which completely duplicates an existing diskette. If not, you should follow these steps:

1) Format a diskette using the NEWDISK command and place it in drive #1.
2) Enter the command "+++GET 0.COPY.CMD".
3) Enter the command "+++MON 0 1", to return to your system monitor.
4) Remove your system disk from drive #0 and insert the master Pascal compile disk.
5) Using your system monitor, jump to location $C100 (the start of the copy utility).
6) Upon receipt of the "COPY COMPLETE" message, replace your system disk and you now have a complete copy of the Pascal disk.

Should you have difficulties with any of the above points or procedures, feel free to write or call between 10 and 12 am.

IMPORTANT NOTE

The complete 6809 Native-Code Pascal Compiler package includes a "compiler" module and a "run-time" module. Both of these modules are copyrighted by Technical Systems Consultants, Inc. and may not be transmitted in any form to any user other than the owner. The binary code generated from the user's source by the compiler module is the property of the user and may be freely distributed by itself. Note, however, that the run-time module may not be distributed, even though it is required to execute compiled binary code.

If a user wishes to distribute the run-time module along with his compiled binary module, he must obtain a license. Contact Technical Systems Consultants, Inc. for details.

REMEMBER:

It is illegal to distribute copies of the run-time module without a license!

Pascal User's Manual
Table of Contents

INTRODUCTION

This user's manual is written to explain the particular implementation-dependent details of our Pascal compiler. This manual is in no way a tutorial for the Pascal programming language. If the user needs to first learn Pascal, we direct your attention to Appendix A where we have listed a few reference texts for the Pascal language.

The primary goal in our implementation was to produce fast and efficient object code. This was one of our major reasons for producing 6809 native code as opposed to interpretive "P-code." We have tried to implement as many of the features in the Jensen and Wirth User's Manual [1] as possible while keeping the objective of fast and efficient code in mind. At the time of initial release of this Pascal system, a Pascal standard has not yet been adopted. For purposes of our Pascal system, we have adopted the Jensen and Wirth User Manual specification as a "standard." This is the reason that we include a copy of the Jenson and Wirth User Manual with our Pascal system. Please refer to this manual when you have questions about standard Pascal syntax or semantics. Sample programs included on our Pascal system disk are used with permission of the Springer-Verlag Publishers in New York.

This manual will describe to the user how to compile and run a Pascal program. We will also describe the features that differ between standard Pascal and our Pascal system; these differences will include features not implemented and additional, non-standard features.

---

[1] Kathleen Jensen and Niklaus Wirth, Pascal User Manual and Report, Second Edition (New York: Springer-Verlag, 1974)

HOW TO COMPILE AND RUN PASCAL PROGRAMS

A. Your Pascal System

The disk containing the Pascal system for the FLEX™ (FLEX is a trademark of Technical Systems Consultants, Inc.) Operating System contains the Pascal compiler, the Pascal run-time package and several example programs. The Pascal compiler programs found on your disk should include:

```
PASCAL.CMD
PRUN.CMD
NPASCAL.BIN
NPASS1.BIN
NPASS2.BIN
NPASS3.BIN
NPASS4.BIN
NPASS5.BIN
NP6809.BIN
PREFIX.TXT
PASCDEF.TXT
```

The purpose of each of these system files will be explained below.

PASCAL.CMD - This is the program that invokes the compiler. The compiler is composed of a runtime package and several Pascal programs which are the various passes. This program simply starts execution of the main routine of the compiler, NPASCAL.BIN.

NPASCAL.BIN - This is the binary code for the main routine of the Pascal compiler. It controls the compilation by handling the scanning of the command line parameters and checking options. This file then calls the various compiler passes in order. These various passes are found in the files NPASS1.BIN, NPASS2.BIN, ..., NP6809.BIN.

PRUN.CMD - This file contains the program to initiate execution of an arbitrary Pascal program (other than the compiler itself). This is the command one uses to "run" a compiled Pascal program. It contains all of the code necessary for running a Pascal program on the FLEX Operating System. The runtime package may be trimmed down in size by eliminating some of the routines that a particular user program does not need. The trimming of the runtime package will be discussed in detail in the "ADAPTING YOUR SYSTEM" section.

PREFIX.TXT - This file contains the Pascal declarations of the "standard environment." These are the runtime procedures which are predeclared for you. These procedures include routines to manipulate the dynamic storage heap, text file handlers, floating-point math packages, etc. Please see Appendix B for a

complete listing of this file for further information. This text file is automatically prepended to all Pascal programs as they are compiled; this provides the definition for "standard" routines.

PASCDEF.TXT - This is an assembler LIB file that is needed when the native-code from the compiler is assembled. This file contains various definitions necessary for the interface between the user's program and the runtime system. Please note, this file must reside on the disk that contains the native-code output from the Pascal compiler, usually the working drive; this is the only way that the 6809 Assembler can include these definitions at assembly time.

B. How to compile and run Pascal programs

This Pascal compiler requires a 56K system in order to compile programs; at least 48K of user RAM is necessary. In order to execute a Pascal program, it must first be compiled into an assembly language program. This assembly language program must then be assembled using the standard FLEX assembler. The binary file that is produced can then be executed. The description of this process is detailed below.

The general form of the command to compile a Pascal source program is:

```
+++PASCAL source-file [assembly-file] [+options]
```

The square brackets ([]) enclose optional parameters on the command line; these parameters may or may not be included. Source-file and assembly-file are standard FLEX file names. If no name is given for the assembly-file, the name will default to the source-file name and the default extension of .ASM for ASseMbly-file. The options are any combination of the letters "BLYCNSQ." Each letter has an individual meaning, as below:

- B - Suppress code generation of the compiler.
- L - Generate a source code listing to the terminal.
- Y - Automatically delete the old assembly-file if it exists.
- C - Suppress the automatic runtime value checks.
- N - Keep track of the exact execution line number.
- S - Print additional summary information at end of compilation.
- Q - Quit compilation after the first pass in which an error is detected.

Some of the above options require some additional explanation. The "C" option will disable some of the runtime value checks. This means range checking of scalars or pointers not NIL will not be checked at runtime. However, array subscript bounds and case statement bounds will always be checked at

runtime. The "N" option instructs the compiler to generate the code necessary so at runtime each line number is kept track of. If an error does occur at runtime, then the exact line number may be reported; otherwise, only an approximate line number is reported. The approximate line number is the line number of the procedure declaration that the statement which produced the runtime error resides within. Finally, the "S" option will produce some additional information at the end of compilation. The most valuable of this information may be the number of conditionals that it was able to optimize. The compiler is designed to optimize all conditionals, such as those appearing in IF-THEN-ELSE, WHILE loops, and REPEAT-UNTIL statements.

During the compilation process, two temporary files are created and deleted. These two files will reside on the FLEX working-drive; therefore, it is necessary that free disk space equal to approximately twice the size of the source program exist on the working-drive disk. Realize that if the source program is on the working-drive, and the output of the compiler is directed to the working-drive, then more space on the disk will be needed to accommodate the temporary files and output of the compiler. As the compiler uses the FLEX system and working-drive numbers, these both must be set to a specific number and not to "ALL".

After the compiler has produced an assembly-file, this file must be run on the assembler.

```
+++ASMB assembly-file [binary-file] [+options]
```

Please refer to the 6809 Assembler User's Manual for details on the syntax and options available. Remember, the default assembly-file name from the Pascal compiler has a .ASM extension. Therefore, the assembly-file name used in the ASMB command line should reflect this extension. Also, recall that the definition LIB file, PASCDEF.TXT, must reside on the same disk as the assembly-file.

Finally, the user should be ready to run the error-free, compiled and assembled Pascal program. The command to do this is:

```
+++PRUN binary-file [parameters] [<input-file] [>output-file]
```

The binary-file is the result of the assembly described above. The optional parameters in the command line are character strings, separated by the delimiters blank or comma, which are passed to the user's Pascal program. The details of these parameters will be discussed in the "NON-STANDARD FEATURES" section under "Running another Pascal program." The "<input-file" is a way of making an external file serve as the default Pascal file INPUT. By using the "<" in front of a FLEX file name, such as <DATA.TXT, all characters read from the default Pascal file INPUT will come from that FLEX file, DATA.TXT in this example. Normally, the default Pascal files INPUT and OUTPUT will be

associated with the terminal. Finally, the ">output-file" works the same way that the "<input-file" above works. This time the Pascal file OUTPUT is now "output-file". Any WRITEs to the default OUTPUT will go to "output-file" instead of the terminal.

STANDARD FEATURES NOT SUPPORTED IN OUR PASCAL SYSTEM

In order to point out the major differences between our Pascal system and standard Pascal, we will first examine the features not supported in our Pascal system that can be found in the Pascal User Manual and Report. We shall make direct references to specific pages out of the "User Manual"; so, please make note of these differences in your copy of the "User Manual." While our Pascal compiler will accept either upper or lower case letters in reserved words, in this manual we will use upper case letters in order to make the reserved words, procedures, functions, and special features stand out. However, all functions, procedures, constants and types found in the PREFIX must be spelled exactly as they appear in the PREFIX. That is, uppercase letters must be uppercase and lowercase letters must be lowercase.

On page 9 of the "User Manual" a discussion of identifiers takes place. Our Pascal system will allow the user to have up to 160 characters significant in an identifier name. We also allow both upper and lower case characters and the use of the underscore character (_) as any character except the first.

Page 14 of the "User Manual" states that as long as at least one of the operands is of type real (the other possibly being of type integer) the following operators yield a real value:

```
*   multiply
/   divide (both operands may be integers, but the
           result is always real)
+   add
-   subtract [2]
```

In this version of our Pascal System, we do not support implicit widening of integers to real numbers. This means that in mixed-mode expressions, when an integer x is used as a real number, the function CONV(x) must be used to convert the integer x to a real number. The function CONV(x) is a standard function in our Pascal system. For example, the following program section would be incorrect.

```
VAR x,y: integer;
    z: real;
BEGIN
  z := x / y
END;
```

However, if the assignment statement were changed to:

---

[2] Ibid., p. 14.

```
z := CONV (x) / CONV (y)    ;
```

then, the program section would be correct.

Individual programs do not require the Program heading statement discussed on page 16 of the "User Manual." The PROGRAM statement is included in the PREFIX file and is automatically prepended to every user Pascal program; please look at the complete listing of PREFIX in APPENDIX B for further details. It will be an error if the user includes a PROGRAM statement. In order to communicate with external files other than INPUT and OUTPUT, the user will have access to these external file names as parameters on the command line. The runtime package supports an array of records containing a character string; this array is called PARAM with the character string called ID. The first parameter on the command line can be found in PARAM [1].ID, the second in PARAM [2].ID, etc. The user may use up to five parameters, each consisting of up to 16 characters in length. For example, if one would want to pass the file name "1.DATA.TXT" to the Pascal program, the following command line would be used:

```
+++PRUN binary-file 1.DATA.TXT
```

The user's program would then need to access this command line parameter by referencing the PARAM record array.

```
VAR file_name: array [1 .. 16] of char;
    file_ok: boolean;
BEGIN
  . . .
  IF PARAM [1].ID = file_name THEN file_ok := true;
  . . .
END; . . .
```

The details of the structure of the PARAM record array may be found in the listing of the PREFIX file and discussed further in the "Running another Pascal program" in the "NON-STANDARD FEATURES" section.

On page 16 and page 31 of the "User Manual," the use of labels and GOTO statements is dicussed. The current version of our Pascal system does not implement labels, label declarations or GOTO statements. It was felt that labels were detrimental to structured programming practices; in most cases, the repetitive statements such as WHILE loops can replace the function of labels and GOTO statements. If a user used labels and GOTO's for error exiting purposes, we have provided an ABORT statement to replace them; the ABORT statement halts execution. The ABORT statement is dicussed further in "NON-STANDARD FEATURES."

Page 50 of the "User Manual" discusses sets and set operations. Our Pascal system is able to accommodate up to 128 elements in a set; however, the ordinal values of those elements must be within the range 0 to 127. This means that a set of

integers must not contain numbers less than 0 or greater than 127. It is impossible to have a set of real numbers because of this limitation. Note that a set will easily accommodate a list of ASCII characters.

Standard Pascal allows the user to nest procedure and function declarations. Our implementation of Pascal does not allow the nesting of procedure or function bodies. This means that you cannot declare procedures within other procedures. This also applies to functions. This feature of nesting, although quite nice, is not really necessary. By leaving it out, the Pascal system can be implemented much more efficiently; the code produced is more efficient.

You may not pass procedures or functions as formal parameters to procedures or functions.

Arrays of characters must contain an even number of elements. Again, this is for efficiency reasons.

A current implementation restriction does not allow an element of an array of characters to be passed as an actual parameter which corresponds to a variable formal parameter of type char. For example,

```
VAR ch: char;
    carray: array [1 .. 10] of char;
PROCEDURE test (VAR c: char);
  . . .
BEGIN
  test (carray [2]);
  . . .
```

is incorrect; however, carray[2] may be first assigned to ch, and then ch may be passed to test. Furthermore, if the formal parameter of test, c, was a value type parameter, ie. no VAR, then there would be no problem.

The standard procedure, DISPOSE(p), is not implemented. Instead, we use two procedures to manipulate the runtime heap called MARK and RELEASE with an argument of type integer. The runtime heap is like a stack; the MARK(i) procedure instructs the system to remember the current value of the top of heap pointer. After the user makes several calls to NEW the top of heap pointer is moved to accommodate the new, dynamically allocated variables. However, if a RELEASE(i) is called, the top of heap pointer is moved to the old "marked" location of the top of heap. This, in effect, frees up the heap of unnecessary, dynamically allocated variables. MARK and RELEASE are further discussed in "Non-standard procedures and functions" in the next section.

The standard procedures PACK and UNPACK are not supported; however, PACKED arrays and records are allowed.

Empty field lists for variant records (page 46 of the User Manual) must be left out; using a pair of matching parentheses, "()", will cause an error. The compiler will not check for all cases of a variant record; therefore, leaving the empty case out of the declaration will not cause an error. Finally, variant records must contain a tag field; you cannot leave this out.

The standard procedures EOF and EOLN both require the explicit file name even if the file is INPUT. For example, EOF(INPUT) or EOLN(DATA).

NON-STANDARD FEATURES

A. Non-standard features

There are several features in our Pascal system that will differ from the standard specification of Jensen and Wirth. This includes the internal representation of string constants, parameter passing, hexadecimal constants, etc. We will discuss each of these non-standard features in detail below.

Comments in Pascal source programs may be enclosed in the standard "{}" pair, the "(* *)" pair or by a pair of double quotes. For example,

```
{ this is a comment}
(* this is also a comment *)
" finally, this is the third way of commenting "
```

Comments may not be nested by using the same start-of-comment character. That is, (* comment1 (* comment2 *) *) is illegal; however, (* comment1 " comment2 " *) is allowed.

Character strings which are written explicitly in Pascal programs contain a null character at the end of each string. This means that the expression 'abc' is an array of 4 characters; the last character is the null character. This can have added significance; comparing an array of characters to an explicitly written string must take this last null character into consideration. For example:

```
BEGIN
  . . .
  IF PARAM [1].ID = '1.PASCALPG.TXT ' THEN . . .
```

Recall that PARAM [x].ID is an array of 16 characters (see APPENDIX B). The explicitly written string, '1.PASCALPG.TXT ', is 15 characters long plus the one last null character; therefore, making a total of 16 characters for the string. If the user did not use 15 characters for the explicitly written string, the compiler would issue an error message; the two expressions must match in type. For example:

```
'1.TEST.TXT     '
 123456789012345
```

must be padded with blanks so that the string is a total of 15 characters in length. If we were comparing an explicitly written string to an array of only ten characters long, then the explicitly written string must contain nine characters. For example:

```
VAR list: array [1 .. 10] of char;
BEGIN
```

```
. . .
IF list = '123456789' THEN . . .
```

Non-scalar data values are always passed by reference when they appear as actual parameters. Formal parameters that are declared as 'value' parameters may not be modified; this is checked at compile time.

The RESET and REWRITE statements have been extended to allow a 'file-name' to be associated with internal Pascal files. The 'file-name' is normally a disk file, in the standard FLEX notation. For example:

```
VAR f: file of integer;
BEGIN
  REWRITE (f, PARAM [2].ID);
                or
  RESET (f, 'INDATA.TXT');
  . . .
```

For text files, the file name may be 'ME'. This implies that all I/O to that file should be directed to the user's terminal. In this case, the programmer should perform a RESET before doing any I/O on the terminal. For example:

```
VAR text_file: TEXT;
BEGIN
  RESET (text_file, 'ME');
  . . .
```

This will associate the text file with the terminal. The procedure REWRITE should not be used on any file associated with the terminal.

All files should be declared in the global block; files that are local to procedures or functions are not supported.

The range of integers is -32768 to +32767, using 16 bits. The range of real numbers is 1.0E-38 to 1.0E+38 and -1.0E-38 to -1.0E+38. Real numbers in our Pascal system also have 16.8 digits of accuracy. All of the ASCII characters from 0 to 127 may be used in our Pascal system.

Hexadecimal constants may appear in Pascal programs by typing them with a "$" preceding. For example, $100 is the decimal value 256. These values are restricted to an integer (16 bits) value; $FFFF is the largest hex value allowed.

A non-printable or printable character may be placed within character strings by enclosing its decimal value (x) within (: :), such as (:x:). For example, the string 'Hi there(:33:)' is the same string as 'Hi there!'.

Both PACKED arrays and PACKED records are implemented; however, because of the word size, PACKED does not change the internal representation. The standard procedures PACK and UNPACK are not implemented.

Pascal type ALFA has been included as a standard type. Its description is an array of ten character elements. That is:

```
TYPE ALFA = array [1 .. 10] of char;
```

The procedure WRITE and WRITELN cannot have the functions ABS or SQR as arguments, such as:

```
WRITE (ABS (x)); WRITELN (SQR (value));
```

Procedures WRITE and WRITELN must know the type of the result (real or integer) in advance. The results of ABS and SQR depend upon the type of the argument. That is, a real number argument will return a real number result; an integer argument returns an integer result.

B. Non-standard procedures and functions

We have included several additional, non-standard procedures and functions to interact with our FLEX Operating System. The Pascal descriptions of most of these subroutines may be found in the PREFIX listing in APPENDIX B. Again, note the exact spelling of the items in the PREFIX.

The heap is handled in a very simple fashion in this Pascal system. When variables are created with the NEW procedure, a pointer to the variable is returned. This pointer is kept in the runtime package and is updated to be the next available memory location for new dynamically allocated variables at any time. If the programmer wishes to return some of these variables to the system, he may do so, but only in a very rudimentary fashion. The current value of the top of the heap may be obtained by using the routine MARK. This will set the integer parameter provided to the current top of the heap. If the user then later calls the routine RELEASE with this same integer value, any memory that was allocated during the time between the MARK(i) and RELEASE(i) will be restored to the system. This means that all variables that were created during this time will be lost. This simple mechanism functions much like a stack for the variables created by NEW.

The routines SYSTEM_DRIVE and WORK_DRIVE simply return the FLEX system or working drive number as a character to the program. This character value may be used for creating file names.

The routine ABORT provides a means for a Pascal program to abnormally terminate. The last line executed as well as the "program aborted" status will be returned to the caller (see "Running another Pascal program"), or printed at the user's terminal if this is the main routine.

The procedures PEEK, POKE, PEEKW and POKEW provide a means for the programmer to access absolute memory locations. PEEK will return the contents of the addressed byte as an integer value (0 to 255). POKE will place the integer value (0 to 255) into the addressed byte. The routines PEEKW and POKEW access words (two bytes) rather than bytes.

Routines __GET through _SREWRITE (see PREFIX listing in APPENDIX B) are used by the system to implement file I/O. Any routine in the PREFIX starting with a "_" or "__" cannot be called by a user's Pascal program.

The procedures BUFFER and UNBUFFER are used with terminal I/O. UNBUFFER will turn the normal buffering mechanism off for a given file which is associated with a terminal. If the file is not attached to a terminal, this routine has no effect. This is used when the user does not want to have to enter an entire line (including the carriage return), as in the case of reading a

single character response from the terminal. BUFFER restores the normal buffering.

The SETBIN procedure sets the FLEX space compression flag for FLEX binary files. This is useful for processing files that you need to read in byte by byte (actually character by character) without treating them like FLEX text files. The Pascal file name is the argument to this procedure.

The function RND has been included as a random number generator. This function returns a random number that has a value between zero and one. The programmer can use this to generate random numbers between any desired limits using the formula:

```
Random_Number := (ML - MS) * RND (0.0) + MS;
```

Where ML is the upper limit and MS is the lower limit. The resulting number that is generated will range from MS to ML. The argument X has an effect on the number that is generated according to the following rules. X must be a real number; the result of RND will also be a real number.

X<0.0 A new series of random numbers is started. For different negative values of X, a different sequence is started each time, but if the argument retains the same value, the function will keep starting the same random sequence so the value returned will be the same each time the function is called.

X=0.0 Causes the function to generate a new random number when it is called. This is the argument that will normally be used with the RND function.

X>0.0 This returns the last random number that was generated.

C. Running another Pascal program

Our Pascal runtime system supports the calling of other Pascal programs or assembly language programs as subroutines. The routine RUN provides a means whereby a Pascal program may run another Pascal program as if calling a subroutine. This routine loads the called Pascal program and then executes it. The user may pass a set of parameters (the PARAM record array) to the called Pascal program. For example, when a Pascal program is executed by the PRUN command, it is executed by the RUN procedure. When the called program terminates, its termination status and last line executed are returned to the calling program. In this fashion, the calling program can be made aware of how things went with the program it called, even if it aborts. For example:

```
VAR prgm, data_file: IDENTIFIER;
    parms: ARGLIST; (* types IDENTIFIER and ARGLIST defined
                       in the PREFIX *)
    line_no: integer;
    reason: PROGRESULT; (* defined in PREFIX *)
 . . .
BEGIN
  prgm := 'QSORT          '; (* call program 'QSORT' *)
  data_file := '1.DATA.TXT     ';(* name of a data file *)
  parms [1].ID := data_file; (* parameter list *)
  RUN (prgm, parms, line_no, reason);

  (* check for any errors *)

  IF reason = TERMINATED THEN(* normal termination: okay *)
  . . .
```

These statements will cause the runtime package to load the program "QSORT.BIN" from the working disk and start its execution. A program is always loaded from a disk file with the default extension of .BIN from the working disk. All files opened by the calling program remain open and may be accessed by the called program. This includes the files INPUT and OUTPUT. Furthermore, all files opened by a called program will be closed when it exits. A called program may call other programs. A program may pass to another program parameters of type IDENTIFIER, integer, or boolean. IDENTIFIER is an array of 16 characters. Obviously, the ability to call other programs allows the users to build up a library of frequently used routines and call them when needed; this allows users to "link" to other Pascal or assembly language programs. Calling other Pascal programs from Pascal programs is not difficult, but the user should have a very good understanding of Pascal and this system before attempting it.

Calling an assembly language routine is very similar to calling another Pascal program; however, the user should have a very good understanding of the system before attempting this.

The user's assembly language program must meet four requirements. The program must be position-independent; therefore, variables should be referenced off of a stack pointer. The program must return the original values of registers U and Y back to the calling program; it is a good idea to first push these two registers onto the stack. The program should have its first two bytes contain the total number of bytes, including the first two, that this program will contain. In other words, the length in bytes of the total program must precede the first executable statement; the third byte of the program is assumed to be the first executable opcode. And, finally, the program must clean up the stack at its conclusion. This would include pulling off the U and Y register if they were pushed and pulling off other system information that Pascal expects pulled off as explained in the following paragraph.

The first two requirements are simple. The third requirement means that an FDB psuedo-op with the total byte count should be placed as the first and second byte of the program. The fourth requirement is accomplished by making a call to routines included in the runtime package. A skeletal example of an assembly language program follows.

```
        FDB     SIZE    size in bytes of the program
START   EQU     $0003   address of the start routine in runtime
TERM    EQU     $0009   address of termination routine in runtime
BEGIN   JSR     START   set up stack
        FDB     0,0,0,0 necessary for the start routine
        PSHS    U,Y     save U and Y registers
*
* Now the the address of the INPUT FCB, OUTPUT FCB and
* the parameter list may be referenced off of the Y register.
*
*     16,Y  --  address of the INPUT FILE FCB
*     14,Y  --  address of the OUTPUT FILE FCB
*     12,Y  --  address of the start of the parameter list
*
  . . . program . . .
        PULS    U,Y     restore registers U and Y
        JSR     TERM    terminate the program
SIZE    EQU     *       size of the program
        END
```

A diagram of the stack after a Pascal or assembly language program is called can be found in APPENDIX C. Notice that the addresses of the FCB for both INPUT and OUTPUT are on the stack. Also note the address of the start of the parameters is also on the stack. All of these addresses may be accessed from register Y as noted above.

ADAPTING YOUR SYSTEM

This section includes the means to trim the runtime package in order for some Pascal programs to run that, due to their size or flow of control, would not run under the normal runtime environment. For example, some programs may not use any real number math but require a great deal of memory to run; we can trim the runtime package down so that this program may be able to run.

The runtime package is organized so that unwanted routines may be overlayed with user program space very simply. There are two types of routines that occur in the runtime package: routines that are called implicitly by Pascal programs, and those that are called explicitly by the user's program. The first type of runtime routine must always be present for any Pascal program to function properly. The second type of routine, explicitly called, may include many routines that are not used by a given Pascal applications environment, such as real number math. In this case, those routines that are not needed may be removed from the runtime package by making some simple patches. This will also make more memory space available to the Pascal program.

The total runtime package is organized into sections. Each section is less important in function than the ones loaded in memory lower than itself. In this fashion, the sections that are most likely to not be needed may be removed and their memory space reused by making this area available to the Pascal program.

Memory in a running Pascal program is set up in two separate areas, the stack and the heap. All normal variables and programs are placed on the stack, which grows downward from the top of available memory. The stack is able to grow downward to the top of the heap, just as the heap is able to grow upward to the top of the stack. Dynamic variables, those variables that are created by calls to the procedure NEW, are placed in the heap which grows up toward the top of memory. The runtime package itself is placed in the lowest memory, starting at location zero. If the start-of-the-heap pointer is set up so that unused runtime routines are overwritten by the heap or possibly the stack in some programs, then that memory will have been reused effectively.

In order to make the necessary modifications to decrease the size of the runtime package, the applications environment must be examined. Those routines that are provided in the runtime package but are not used, may be considered candidates for deletion. To make the process simpler, the routines are organized in memory in the same fashion that their declarations appear in the standard PREFIX. These declarations map directly onto an address table found in the runtime package at location $100 (hex 100). Each routine has a one word (two byte) entry in

the address table. Therefore, the address of the sixth routine in the PREFIX declarations is at $10A ($100 + (6 - 1) * 2). If routines six and up were found to be unnecessary, all that would be needed to reuse their memory space is to set the start-of-the-heap pointer (locations $180 and $181) to the address found in locations $10A and $10B. It should be noted that the runtime package may only be trimmed back to the first used routine; that is, the user cannot eliminate some of the real number math routines and a few of the file I/O routines. The memory space that is reused must be contiguous. Setting the start-of-the-heap pointer back reclaims all memory from that address through the end of the runtime package for use by the application program.

For example, let's trim off the floating point trigonometric, exponential, square root and random number generator routines from the runtime package. These are routines numbered 39 to 45 (see APPENDIX B). Therefore, the address of the first trig function can be found in addresses $014C and $014D ($100 + (39 - 1) * 2). Next, we would set the start-of-the-heap pointer (addresses $0180 and $0181) to point to the address found in locations $014C and $014D. For this example only, let's say that $0400 was the contents of locations $014C and $014D. This means that we should put $0400 in locations $0180 and $0181, the start-of-the-heap pointer. We have now reused the memory from the old start-of-the-heap down to location $0400.

It is a good idea to set all of the entries in the address table to zero for the routines that have been removed. In the example above, locations $014C through $0159 inclusive should be set to zero. This is an illegal value and will cause a program abort if a routine that has been removed in this fashion is inadvertently called. It is not advised to remove the declarations from the PREFIX.

APPENDIX A.

Additional references for Pascal

If the user feels that additional information is needed to help supplement this manual and the "User Manual", we have listed a few reference texts to the Pascal programming language. This list is in no way an exhaustive list of references; furthermore, we do not discredit any references not found in this list.

Conway, R., Gries, D. and Zimmerman, E.C. [1976] A Primer On Pascal, Winthrop, 1976.

Findlay, W. and Watt, D.A. [1978] Pascal, An Introduction to Methodical Programming, Computer Science Press, Inc., 1978.

Grogono, Peter [1978] Programming in Pascal, Addison Wesley, 1978.

Webster, C.A.G. [1976] Introduction to Pascal, Heyden and Son, 1976.

Wilson, I.R. and Addyman, A.M. [1978] A Practical Introduction to Pascal, Springer-Verlag, 1978.

Wirth, N. [1973] Systematic Programming - An Introduction, Prentice-Hall, 1973.

Wirth, N. [1976] Algorithms + Data Sturctures = Programs, Prentice-Hall, 1976.

In order to receive more information about the new developments in the Pascal programming language, you may want to subscribe to Pascal News. This is the official publication of the Pascal User's Group (PUG). It contains letters, articles and implementation notes. The address to write to and receive more information is:

Pascal User's Group
University Computer Center: 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455
USA

APPENDIX B.

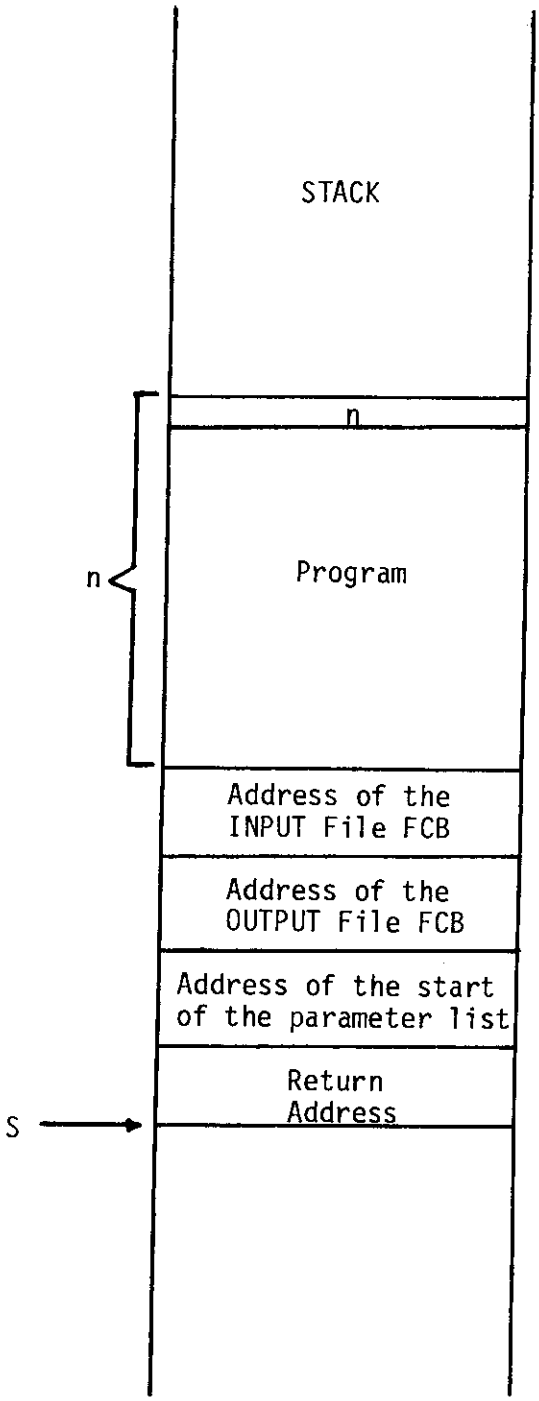
```
(*************************************************************************)
(*                    S T A N D A R D    P R E F I X                     *)
(*************************************************************************)

  CONST LINELENGTH = 132; { maximum line length }
  TYPE LINE = ARRAY [1..LINELENGTH] OF CHAR;

  CONST IDLENGTH = 16; { maximum length of an identifier parameter }
  TYPE IDENTIFIER = ARRAY [1..IDLENGTH] OF CHAR;

  TYPE
    TEXT = FILE OF CHAR;
    text = TEXT;

  CONST MAXSTR = 10; { LENGTH OF ALFA STRING }
  TYPE
    ALFA = ARRAY [1 .. MAXSTR] OF CHAR;
    alfa = ALFA;

  TYPE PROGRESULT =
    (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,
     HEAPLIMIT, STACKLIMIT, ABORTED);

  TYPE ARGTAG =
    (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE);

  TYPE ARGTYPE = RECORD
                   CASE TAG: ARGTAG OF
                     NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
                     INTTYPE: (INT: INTEGER);
                     IDTYPE: (ID: IDENTIFIER)
                 END;

  CONST MAXARG = 5; { maximum number of arguments passed to a program }
  TYPE ARGLIST = ARRAY [1..MAXARG] OF ARGTYPE;

  { Heap manipulation routines }

1 PROCEDURE MARK(VAR TOP: INTEGER);
2 PROCEDURE RELEASE(TOP: INTEGER);

  { To call another Pascal or assembly language program }

3 PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;
                VAR LINE: INTEGER; VAR RESULT: PROGRESULT);

4 PROCEDURE SYSTEM_DRIVE(VAR C: CHAR); { return '0' or '1' }
5 PROCEDURE WORK_DRIVE(VAR C: CHAR); { return '0' or '1' }

6 PROCEDURE ABORT;  "TERMINATE PROGRAM IMMEDIATELY"
```

```
   { PROCEDURES USED TO MANIPULATE ABSOLUTE MEMORY LOCATIONS. }

 7 FUNCTION PEEK(LOC: INTEGER): INTEGER;  (* READ BYTE *)
 8 FUNCTION PEEKW(LOC: INTEGER): INTEGER; (* READ WORD *)

 9 PROCEDURE POKE(LOC, VAL: INTEGER);  (* WRITE BYTE *)
10 PROCEDURE POKEW(LOC, VAL: INTEGER); (* WRITE WORD *)

   { Standard functions }

11 FUNCTION ODD(X: INTEGER): BOOLEAN;
12 FUNCTION ROUND(X: REAL): INTEGER;

(*********************************************************************)
(*                      FILE I/O DEFINITIONS                         *)
(*********************************************************************)

   {        ROUTINES STARTING WITH '_' OR '__' ARE NOT DIRECTLY
                      ACCESSABLE TO USERS!                           }

13 PROCEDURE __GET(VAR F: TEXT);
14 PROCEDURE __PUT(VAR F: TEXT);

15 PROCEDURE __RDX(VAR C: CHAR);
16 PROCEDURE __WRX(C: CHAR);

17 FUNCTION EOLN(VAR F: TEXT): BOOLEAN;
18 FUNCTION EOF(VAR F: TEXT): BOOLEAN;

19 PROCEDURE __RLN;
20 PROCEDURE __WLN;

21 PROCEDURE __RWF(VAR F: TEXT;  DUMMY1, DUMMY2: INTEGER);
22 PROCEDURE __RWFS(VAR F: TEXT);  { SHORT FORM OF RWF }
23 PROCEDURE __EIO;

24 PROCEDURE __RDI(VAR I: INTEGER; WIDTH, DIGITS: INTEGER);
25 PROCEDURE __RDC(VAR C: CHAR; WIDTH, DIGITS: INTEGER);
26 PROCEDURE __RDR(VAR R: REAL; WIDTH, DIGITS: INTEGER);

27 PROCEDURE __WRI(I: INTEGER; WIDTH, DUMMY: INTEGER);
28 PROCEDURE __WRC(C: CHAR; WIDTH, DUMMY: INTEGER);
29 PROCEDURE __WRS(S: LINE; WIDTH, DUMMY: INTEGER);
30 PROCEDURE __WRR(R: REAL; WIDTH, DIGITS: INTEGER);

31 PROCEDURE _FRESET(SIZE: INTEGER; NAME: LINE);
32 PROCEDURE _FREWRITE(SIZE: INTEGER; NAME: LINE);

33 PROCEDURE _SRESET;
34 PROCEDURE _SREWRITE;

35 PROCEDURE BUFFER(VAR F: TEXT);  (* TURN BUFFERING ON *)
36 PROCEDURE UNBUFFER(VAR F: TEXT);  { SINGLE CHARACTER TERMINAL INPUT }
```

```
37 PROCEDURE PAGE(VAR F: TEXT);  (* OUTPUT FORM FEED *)
38 PROCEDURE SETBIN(VAR F: TEXT);  { SET BINARY FILE MODE }

(*************************************************************)
(*  STANDARD FUNCTIONS (TRIG, ETC)                           *)
(*************************************************************)

39 FUNCTION SIN(X: REAL): REAL;
40 FUNCTION COS(X: REAL): REAL;
41 FUNCTION ARCTAN(X: REAL): REAL;

42 FUNCTION EXP(X: REAL): REAL;
43 FUNCTION LN(X: REAL): REAL;

44 FUNCTION SQRT(X: REAL): REAL;
45 FUNCTION RND(X: REAL): REAL;

   PROGRAM P(VAR INPUT, OUTPUT: TEXT;  VAR PARAM: ARGLIST);
```

APPENDIX C.

Diagram of stack after a program is called



APPENDIX D.

Example programs on Pascal System disk

Several programs from the Pascal User Manual were included on the Pascal System disk. Please examine these sample programs and use them as a template for your own development of programs under this Pascal system. We have also included another Pascal program called XREF.TXT that produces a cross-reference listing of Pascal programs. This program was modified from Algorithms + Data Structures = Programs, by Niklaus Wirth published by Prentice Hall in New Jersey. Please examine it to see how the command line is accessed to get the input file name and option. The user may use this program by typing:

```
+++PRUN XREF Pascal-file-name [+L]
```

The Pascal-file-name is the FLEX file-name for the source program written in Pascal; the option +L is for suppressing the listing of the source program. Try running a cross-reference on the cross-reference program, itself. Please note that all of the programs are in source form; the user must first compile these programs before running them. The programs included on the disk are:

```
COSINE.TXT
GRAPH2.TXT
COMPLEX.TXT
SETOP.TXT
PRIMES.TXT
MINMAX3.TXT
TRAVERSL.TXT
EXPON2.TXT
RECURGCD.TXT
XREF.TXT
```