UniFLEX™ Pascal User's Manual

technical systems consultants, inc.

UniFLEX™ Pascal User's Manual

Technical Systems Consultants, Inc.
P.O. Box 2570
West Lafayette, Indiana 47906

MANUAL REVISION HISTORY

Revision	Date	Change
A	4/81	Original Release, Pascal System Version 1.0
B	9/81	change page numbering page 4-6: added description on "lock" and "unlock"

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.

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.

Pascal User's Manual
Table of Contents

1. INTRODUCTION 1-1

2. HOW TO COMPILE AND RUN PASCAL PROGRAMS
   A. Your Pascal System 2-1
   B. How to compile and run Pascal programs 2-3

3. STANDARD FEATURES NOT SUPPORTED BY OUR PASCAL SYSTEM 3-1

4. NON-STANDARD FEATURES
   A. Non-standard features 4-1
   B. Non-standard procedures and functions 4-4
   C. Running another Pascal program 4-9

5. APPENDIX
   A. Additional references for Pascal A-1
   B. Listing of the standard PREFIX file B-1
   C. Listing of the system PREFIX file C-1
   D. Example programs on Pascal System disk D-1

IMPORTANT NOTE

The complete 6809 Native-Code Pascal Compiler package includes a "compiler" module and two "run-time" modules. All 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 modules may not be distributed, even though one is required to execute compiled binary code.

If a user wishes to distribute a 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 modules without a license!

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 UniFLEX™ (UniFLEX is a trademark of Technical Systems Consultants, Inc.) Operating System contains the Pascal compiler, the Pascal run-time packages and several example programs. The Pascal compiler programs found in your system should include:

```
/bin/pascal
/bin/pascal.passes/np6809
/bin/pascal.passes/npass1
/bin/pascal.passes/npass2
/bin/pascal.passes/npass3
/bin/pascal.passes/npass4
/bin/pascal.passes/npass5
/bin/pascal.passes/pascal
/bin/pascal.passes/prefix
/bin/pascal.passes/sysprefx
/bin/nrun
/bin/sys_rt
/gen/errors/system
/lib/pascdef
```

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

/bin/pascal - 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, /bin/pascal.passes/pascal.

/bin/pascal.passes/pascal - 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 /bin/pascal.passes/npass1, /bin/pascal.passes/npass2, ..., /bin/pascal.passes/np6809.

/bin/pascal.passes/prefix - 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.

/bin/pascal.passes/sysprefx - This file contains the Pascal

declarations of the "system environment". These are the runtime procedures that are predeclared in the same way as the standard prefix above. These procedures allow the user to make calls directly to the UniFLEX Operating System routines. This includes changing permissions on files, linking files, creating files, etc. This prefix does not include any of the floating point procedures. Please examine APPENDIX C for details. Only one of the prefixes may be used for a specific program.

/bin/nrun and /bin/sys_rt - These are the runtime packages for the standard runtime environment and system environment. These files are used by the shell program for Pascal program execution and normally not directly used by the user.

/gen/errors/system - This file contains various pascal run-time error messages. These error messages are obtained and displayed by the appropriate run-time package during execution. Note, this file is not a text file; it cannot be listed.

/lib/pascdef - This file contains various assembly equates necessary for the assembler to assemble the code produced by the Pascal compiler.

B. How to compile and run Pascal programs

In order to execute a Pascal program, it must first be compiled into an assembly language program. The assembly language program must then be assembled using the standard UniFLEX 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 sourcefile [resultfile] [+options]
```

Sourcefile is a standard UniFLEX file name containing the Pascal source program to be compiled. Only one source file may be compiled; the compiler will issue an error message if the user specifies more than one Pascal source file. The optional resultfile is also a standard UniFLEX file name that will contain the result of the compilation process. This result may be a file of assembly language mnemonics or the final executable binary code, as described below. If the sourcefile name ends with a ".p" extension, Pascal will compile the program and, if there are no errors, will automatically enter the assembler and produce the executable binary code. This code will be placed in the resultfile if it is specified, or in a file with the same name as the sourcefile name without the .p extension if the resultfile is not specified. That is,

```
++ pascal test.p
```

will produce a binary file called "test". However,

```
++ pascal test.p /bin/testprgm
```

will produce a binary file called "/bin/testprgm". If a sourcefile name does not include a .p extension, then a file containing the assembly language mnemonics will be produced. If a resultfile was specified, the assembly language mnemonics will be found in the resultfile. However, if no resultfile was specified, the mnemonics will be stored in a file with a name comprised of the sourcefile name and a ".a" extension. For example,

```
++ pascal test /usr/bin/tmp
```

will produce a file of assembly language mnemonics in the file "/usr/bin/tmp". Furthermore,

```
++ pascal test
```

will produce a file of assembly language mnemonics in "test.a". It should be noted that any existing output file will be automatically deleted by Pascal. That is, if "test.a" existed before, it would have been deleted, and a new "test.a" file would

have been created. The options above are any combination of the letters "ablcnqu." Each letter has an individual meaning, as below:

a - Suppress execution of the assembler for sourcefile names with a .p extension.
b - Suppress code generation of the compiler.
l - Generate a source code listing to standard output.
c - Suppress the automatic runtime value checks.
n - Keep track of the exact execution line number.
q - Quit compilation after the first pass in which an error is detected, except pass one.
u - Use the system prefix instead of the standard prefix.

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 that each line number is kept track of at runtime. 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.

After the compiler has produced a resultfile, this file must be run on the assembler if it is a file of assembly language mnemonics. We will call this the assemblyfile.

```
++ asmb [+options] assemblyfile
```

Please refer to the UniFLEX 6809 Assembler User's Manual for details on the syntax and options available. If a resultfile was specified in the "pascal" command, this file will be the assemblyfile. However, if the resultfile was not specified above, the default assemblyfile name from the Pascal compiler is the sourcefile name with a .a extension. Therefore, the assemblyfile name used in the asmb command line should reflect this extension. Furthermore, if the user does assemble the program, the execute permission must be set using the perms command in UniFLEX.

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

```
++ binaryfile [parameters]
```

The binaryfile is the result of the assembly described above. However, this binaryfile name may be a result from compiling a source file with a .p extension. In this case the binaryfile

name will be the sourcefile name without the .p extension. 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 "STANDARD FEATURES NOT SUPPORTED IN OUR PASCAL" section and the "NON-STANDARD FEATURES" section under "Running another Pascal program."

A few examples may help.

```
++ pascal test5 +lcyq
++ asmb test5.a +lso=testcase
++ perms testcase o+x
++ testcase

++ pascal qsort.p
++ qsort </data >sorted
```

The first example compiles a Pascal program named test5 using options described above. The output of the compiler is then assembled producing a binary file called testcase. Now, the execute permission must be set, then this file may be executed by typing its name directly. The second example demonstrates the use of the .p extension. Here the assembly step is done automatically if the compiler discovers no errors in the source program. This program is then executed by getting its input from a file called /data and putting the output in a file called sorted. The second example is much easier to use; it is suggested that the programmer uses the .p extension.

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 all upper or lower case letters in reserved words, in this manual we will use upper case letters in order to make the reserved words stand out. Again, the user may use lower case letters to spell these words.

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:

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

then, the program section would be correct.

---

[2] Ibid., p. 14.

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 /bin/pascal.passes/prefix in APPENDIX B and /bin/pascal.passes/sysprefix in APPENDIX C for further details. It will be an error if the user includes a PROGRAM statement.

Normally, the links between internal Pascal file names and external files are declared in the PROGRAM statement; however, without that statement the user must use another method. 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 pointers pointing to the parameter names found on the command line, including the program name itself. An array of arguments, called "param", contains a variant record with a tag field and a record called "parms" for command line parameters. For command line parameters we will only be interested in the first element of param, that is, param [1]. More information about the param array will be discussed in the "NON-STANDARD FEATURES" section under "Running another Pascal program". The record parms contains the count of parameters on the command line including the program name and a pointer to the array of pointers. For example, the count of parameters, called "arg_count," for:

```
++ test +l
```

is two; the program name "test" counts as one and the "+l" counts as the second parameter. Furthermore, arg_count for:

```
++ test +lsc data
```

is three, "test", "+lsc" and "data". It should be noted that the param [1].parms.arg_count for all programs must be at least one, the program name itself. The pointer, param [1].parms.args, points to an array of pointers. The first subscript of this array is zero; the maximum number of parameters from the command line is 101 including the program name. The first element of this array, param [1].parms.args^ [0], points to an array of characters containing the characters that make up the program name itself. That is, if the user typed the following command line:

```
++ qsort data/testdata +le
```

param [1].parms.args^ [0]^ would equal the characters: "qsort"; up to a total of 64 characters. Furthermore, param [1].parms.args^ [1]^ equals "data/testdata"; while param [1].parms.args^ [2]^ equals "+le". As you may have noticed each command line parameter may be accessed in this fashion. Each parameter may consist of up to 64 characters in length. For example, if one would want to pass the file name "/data" to the

Pascal program, the following command line would be used:

```
++ binaryfile /data
```

The user's program would then need to access this command line parameter by referencing the param structure.

```
VAR file_name: array [1 .. 64] of char;
    file_ok: boolean;
BEGIN
  . . .
  IF param [1].parms.args^ [1]^ = file_name
     THEN file_ok := true;
  . . .
END; . . .
```

The details of the param structure may be found in the listings of the prefix files. It should be noted that this structure is the standard UniFLEX command line parameter passing structure; a brief description of this structure may be found under the description of the exec system call in the "Introduction to UniFLEX™ System Calls" section of the UniFLEX manual.

On page 16 and page 31 of the "User Manual," the use of labels and GOTO statements is discussed. 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 programmer 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 discussed 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 the variant record; therefore, leaving the empty case out of the declaration will not cause an error. Finally, variant records must contain a tag field.

The standard procedures eof and eoln both require the explicit file name even if the file is input.

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.

Pascal source files may be included in other Pascal source files for compilation purposes by using a "%" in column one immediately followed by the file name immediately followed by a carriage return. For example, if there is a file of variable declarations common to several Pascal programs called "declare", these declarations may be included in a source file at compilation time by putting "%declare" starting in column one.

```
%declare
BEGIN
  . . .
END.
```

Nothing may follow the file name except a carriage return. The specified file is first searched for in the current directory, then the "current-directory/plib" and, finally, the directory "/plib". Include files may be nested; that is, a file that has been included within a Pascal program may also include other files.

Comments may be enclosed by using one of three delimiter character sets. The standard curly bracket '§ †' pair, the '(* *)' pair and a pair of double quotes '" "' may be used for comments within a Pascal program. Comments normally may not be nested, ie. "Comment "next" ". However, "Comment (*next*) " is legal.

Tab characters may be read by the Pascal compiler as valid characters. However, when a listing is produced by the compiler, a single space is printed instead of the tab.

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:

```
TYPE file_names = array [1..14] of char;
VAR file_array: array [1..5] of file_names;
BEGIN
  . . .
  IF file_array [2] = 'datafile/data' THEN . . .
```

The array, file_array is an array of 14 characters. The explicitly written string, "datafile/data", is 13 characters long plus the one last null character; therefore, making a total of 14 characters for the string. If the user did not use 13 characters for the explicitly written string, the compiler would issue an error message; the two expressions must match in type. For example:

```
'test/test1    '
 1234567890123
```

must be padded with blanks so that the string is a total of 13 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 UniFLEX notation. The REWRITE statement will not set EOF. For example:

```
VAR f: file of integer;
BEGIN
  REWRITE (f, param [1].parms.args^ [3]^);
                or
  RESET (f, 'indata');
  . . .
```

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.

While READ and WRITE will access UniFLEX standard input and output, "message" was added to write to standard error, the terminal, in the same way. Message should be used whenever an error message needs to be reported. For example,

```
message ('Illegal option specified!');
```

will print "Illegal option specified!" on the user's terminal even if the standard output is being redirected to a file or piped into another program.

B. Non-standard procedures and functions

We have included several additional, non-standard procedures and functions to interact with our UniFLEX Operating System. The Pascal descriptions of most of these subroutines may be found in the /bin/pascal.passes/prefix or /bin/pascal.passes/sysprefix listings in APPENDIX B and APPENDIX C. Both the standard runtime and system runtime environments support some of the same routines. It is very important that the name of the procedure, function or data type is spelled exactly how it is found in the prefixes; this includes upper and lower case letters. The user should consult the listing of the prefix for the environment that they will be running in for a complete list of routines included. However, all of the routines will be described below with the standard runtime routines discussed first. The routines associated with just the system runtime environment will be discussed later.

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 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.

Routines __GET through _SREWRITE (see PREFIX listing in APPENDIX B) are used by the system to implement file I/O. Any routine in either 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 procedure "flush" works for output files opened for writing that are associated to a terminal. Output to a file is normally buffered; when the buffer becomes full, the buffer is written to the file. The flush procedure will flush the buffer at that time instead of waiting until the buffer becomes full.

Just as the kill command in UniFLEX deletes a UniFLEX file, the procedure "kill" deletes a UniFLEX file. The permissions are checked before the actual deletion.

Under the UniFLEX version of Pascal, the user is allowed to seek to random positions in a UniFLEX file. This may be accomplished by using the procedure "seek" with the Pascal file name, the item number, and whether you wish to get or put the item as arguments. The item in this case is whatever type the file was declared to be made up of. That is, if the file is declared to be a file of arrays, then the item is an array, and the item number is the nth array in that file of arrays. If the user wishes to get an item, "Get" is used; if the user wishes to put an item, "Put" is used. Random files begin with record zero (0). External random files opened for update must first use a RESET statement and then a REWRITE statement with just the internal name. For example,

```
VAR f : file of integer;
    input_number : integer;
  BEGIN
    RESET (f, param [1].parms.args^ [2]^);
    REWRITE (f); (* set update permissions *)
    seek (f, 3, Get); (* Get the 4th integer in file f *)
    input_number := f^; (* put integer into variable *)
    f^ := 24;
    seek (f, 3, Put); (* replace 4th integer with 24 *)
   . . .
```

Note that the input and output item is pointed at by the file buffer pointer. Once EOF is set by a "Get", it must be cleared by doing a "Put".

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.

The procedure "system" allows the user to call a utility program in the UniFLEX Operating System from a Pascal program. This allows the calling of such utilities as dir, page, list, etc. For example,

```
system ('dir /bin');
system ('page text');
```

Note that the argument of system must be a string of 64 or less characters enclosed in single quotes. The Pascal program will stop it's execution until the utility has returned back to the sleeping program. If the utility called aborts, then the program will also abort.

The procedures lock and unlock are included to perform record locking on UniFLEX files. Both procedures require the internal Pascal file to be passed, for example "f". Procedure "lock" additionally requires the item number that is to be locked. As in the "seek" procedure, the "lock" procedure numbers items starting at zero and "item" refers to one of whatever elements that comprise the file (viz. arrays, records, etc.). Only one item may be locked at one time. If an attempt is made to lock an item while another item is locked, the first item is automatically unlocked before the second item is locked. The "error" response from the "lock" procedure is "false" if the item was successfully locked. If the response is "true", the item should be assumed to be locked by another task.

The system prefix, /bin/pascal.passes/sysprefx is used when the program wishes to develop system routines to interface directly with the UniFLEX Operating System. Note that the system runtime environment does not support real numbers. The use of this runtime environment should be reserved for those users very familiar with the UniFLEX Operating System and system calls. Please examine the listing of this prefix file in APPENDIX C.

Most of the system procedures listed in the system prefix file can be mapped directly onto system calls described in the "Introduction to UniFLEX™ System Calls" in the UniFLEX manual. This list includes chacc, chdir, chown, chprm, cpint, create,

crtsd, defacc, gtid, guid, link, setpr, status, suid, time, ttime, unlink and update. All of these routines will return a system error if an error occurred and a zero if there was no error. The system error numbers may be found in the system calls section of the UniFLEX manual. These system calls are best described in the UniFLEX manual; however, a description of their syntax in Pascal can be found in the system prefix listing in APPENDIX C.

Procedures fopen, fclose, fread, fwrite and fseek work the same as the system calls open, close, read, write and seek. Procedure "fread" returns the number of characters read in the variable count. Realize, the procedure "fseek" is not like the seek described above. Procedure fseek will seek to specific byte in a file, not to a specific item in the file.

Procedure "sleep" will put the program to sleep for the specified number of seconds. That is, the task is suspended for the specified number of seconds and then activated. A sleep will allow the CPU to operate on other tasks while the task is sleeping; this will help to eliminate the use of "timing loops" in programs.

The procedure "terminate" is similar to the "term" system call. The programmer is able to return the error status back to UniFLEX.

The procedures f_ttyget, f_ttyset and f_ofstat are the same as the system calls ttyget, ttyset and ofstat. However, the Pascal procedures ttyget, ttyset and ofstat are similar to the system calls except that instead of the file descriptor, the actual file is passed as an argument.

Functions "ctoi" and "btoi" convert two charcters to a 2 byte integer and one byte to a signed 2 byte integer, respectfully. Function ctoi expects a character pair starting with the character passed to this function. For example,

```
VAR
  a : alfa;
  b : integer;
  BEGIN
  . . .
    a := 'The rain ';
    b := ctoi (a [2]);
  . . .
```

This will convert the characters "he" in "The" to a two byte integer. The function btoi will take a one byte character and convert it to a signed two byte integer.

Procedures "long_add" through "long_div" perform the math functions of addition, subtraction, multiplication and division upon integers using 4 bytes instead of 2. However, the division

is performed with one long integer as the dividend and one normal (2 byte) integer used as the divisor. The long quotient and integer remainder are returned.

C. Running another Pascal program

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 array) to the called Pascal program. 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. Realize that all of the programs must be able to run under the caller's runtime environment. That is, if the standard runtime environment is in effect, a program called cannot use the routines in the system runtime. For example:

```
VAR data_file: identifier;
    parameters: ARGLIST; (* types identifier and ARGLIST
                            defined in the PREFIX *)
    line_no: integer;
    reason: PROGRESULT; (* defined in PREFIX *)
BEGIN
  . . .
  parameters [1]. tag := IDTYPE; (* set tag type *)
  parameters [1].id := data_file; (* parameter list *)
  run ('qsort', parameters, line_no, reason);
  IF reason = TERMINATED THEN (*normal termination: okay*)
  . . .
```

These statements will cause the runtime package to load the program "qsort" from the working directory and start its execution. 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 64 characters. Furthermore, if users want to use the standard UniFLEX parameter passing structure, they may do so by using the same structure described in the "STANDARD FEATURES NOT SUPPORTED IN OUR PASCAL SYSTEM" section of this manual. Briefly, the user would set up the array of pointers. The name of the program called would be put in parameter [1].parms.args^ [0]^ with the other parameters put in elements args^ [1]^ through n. Furthermore, remember to also set the argument count, parameter [1].parms.arg_count to the proper count including the called program. For example,

```
VAR name: identifier; . . .
PROCEDURE copyid (var result: identifier; in: identifier);
(* copy string to an identifier *)
  BEGIN . . . END; (* not shown *)
 . . .
BEGIN
  . . .
```

```
parameters [1].tag := PARMTYPE; (* set tag *)
new (parameters [1].parms.args); (* allocate pointer *)
parameters [1].parms.arg_count := 3; (* 3 arguments *)
new (parameters [1].parms.args^ [0]);
copyid (name, 'qsort'); (* copy string to identifier *)
parameters [1].parms.args^ [0]^ := name; (* program name *)
new (parameters [1].parms.args^ [1]);
copyid (name, 'data1'); (* copy file name to id *)
parameters [1].parms.args^ [1]^ := name;
new (parameters [1].parms.args^ [2]);
copyid (name, 'data2'); (* file name to id *)
parameters [1].parms.args^ [2]^ := name;
run ('qsort', parameters, line_no, reason);
. . .
```

This brief, general example gives the programmer an idea of what is involved in using this technique. This method of passing parameters is very complex to the novice; therefore, it is suggested that the user have a firm understanding of the system before using this method. 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 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.

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    UniFLEX    P R E F I X                    }
(*************************************************************************)

const idlength = 64;
type identifier = array [1..idlength] of char;

type
  text = file of char;
  alfa = array [1..10] of char;

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

type
  ARGTAG = (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PARMTYPE);

type arg_ptr = ^ arg_list;
     id_ptr = ^ identifier;
     arg_list = array [0..100] of id_ptr;

type UniFlex_args = record
                      arg_count: integer;
                      args: arg_ptr
                    end;

type ARGTYPE = record
                 case tag: ARGTAG of
                   NILTYPE, BOOLTYPE: (BOOL: boolean);
                   INTTYPE: (INT: integer);
                   IDTYPE: (ID: identifier);
                   PARMTYPE: (parms: UniFlex_args)
               end;

const MAXARG = 10;
type ARGLIST = array [1..MAXARG] of ARGTYPE;

type direction = (Get, Put);

 (* Heap manipulation procedures *)

procedure mark(var TOP: integer);
procedure release(TOP: integer);

 (* Call another Pascal program *)

procedure run(ID: identifier; var PARAM: ARGLIST;
              var LINE: integer; var RESULT: PROGRESULT);

procedure abort;  "TERMINATE PROGRAM IMMEDIATELY"
```

```
 (* Standard Pascal functions *)

function odd(X: integer): boolean;
function round(X: real): integer;

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

 (* Routines that begin with "_" or "__" may not be directly
    called by the programmer!  *)

procedure __GET(var F: text);
procedure __PUT(var F: text);

procedure __RDX(var C: char);
procedure __WRX(C: char);

 (* Any type of file may be used as an argument to eoln and eof *)

function eoln(var F: univ text): boolean;
function eof(var F: univ text): boolean;

procedure __RLN;
procedure __WLN;

procedure __RWF(var F: text;  DUMMY1, DUMMY2: integer);
procedure __RWFS(var F: text);  (* SHORT FORM of RWF *)
procedure __EIO;
procedure __SETMSG;

procedure __RDI(var I: integer; WIDTH, DIGITS: integer);
procedure __RDC(var C: char; WIDTH, DIGITS: integer);
procedure __RDR(var R: real; WIDTH, DIGITS: integer);

procedure __WRI(I: integer; WIDTH, DUMMY: integer);
procedure __WRC(C: char; WIDTH, DUMMY: integer);
procedure __WRS(S: identifier; WIDTH, DUMMY: integer);
procedure __WRR(R: real; WIDTH, DIGITS: integer);

procedure _FRESET(SIZE: integer; NAME: identifier);
procedure _FREWRITE(SIZE: integer; NAME: identifier);

procedure _SRESET;
procedure _SREWRITE;

procedure buffer(var F: text);  (* TURN BUFFERING ON *)
procedure unbuffer(var F: text);  (* TURN BUFFERING OFF *)

procedure page(var F: text);  " OUTPUT FORM FEED "
```

```
(*************************************************************)
{  Functions unique to the UniFlex programming environment  }
(*************************************************************)

procedure flush(var f: text);  "Flushes buffered output to 'f'"

procedure kill(which: identifier);  (* Deletes a file *)

(* seek to an absolute position in a file.
 Used by the seek procedure *)

procedure __seek(var f: text;  pos: integer; dir:direction);

(*************************************************************)
{         Standard Functions (trig, etc)                    }
(*************************************************************)

function sin(X: real): real;
function cos(X: real): real;
function arctan(X: real): real;

function exp(X: real): real;
function ln(X: real): real;

function sqrt(X: real): real;
function rnd(X: real): real;

 (* Execute another program in the system *)

procedure system(what: identifier);

procedure lock(var f: univ text;  n: integer;  var error: boolean);
procedure unlock(var f: univ text);

program P(var input, output: text;  var param: ARGLIST);
```

APPENDIX C.

```
(*****************************************************************)
{              UniFLEX  S y s t e m  P r e f i x                  }
{  Use this prefix (via the +u option) for writing systems type   }
{  programs for UniFLEX in Pascal.                                }
(*****************************************************************)

const idlength = 64;
type identifier = array [1..idlength] of char;

type
  text = file of char;
  alfa = array [1..10] of char;

type
  PROGRESULT = (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
                NOREALERROR, HEAPLIMIT, STACKLIMIT, ABORTED);

type
  ARGTAG = (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PARMTYPE);

type arg_ptr = ^ arg_list;
     id_ptr = ^ identifier;
     arg_list = array [0..100] of id_ptr;

type UniFlex_args = record
                      arg_count: integer;
                      args: arg_ptr
                    end;

type ARGTYPE = record
                 case tag: ARGTAG of
                   NILTYPE, BOOLTYPE: (bool: boolean);
                   INTTYPE: (int: integer);
                   IDTYPE: (id: identifier);
                   PARMTYPE: (parms: UniFlex_args)
               end;

const MAXARG = 10;
type ARGLIST = array [1..MAXARG] of ARGTYPE;

 (* TYPES used for seeking *)

type direction = (Get, Put);
type seek_kind = (Start, Current, End);

 (* Define UniFLEX System errors *)

const
  ENONE  =  0;  EIO    =  1;  EFAULT =  2;  EDTOF  =  3;
  ENDR   =  4;  EDFUL  =  5;  ETMFL  =  6;  EBADF  =  7;
  ENOFL  =  8;  EMSDR  =  9;  EPRM   = 10;  EFLX   = 11;
```

```
  EBARG  = 12;  ESEEK  = 13;  EXDEV  = 14;  ENBLK  = 15;
  EBSY   = 16;  ENMNT  = 17;  EBDEV  = 18;  EARGC  = 19;
  EISDR  = 20;  ENOTB  = 21;  EBBIG  = 22;  ESTOF  = 23;
  ENCHD  = 24;  ETMTS  = 25;  EBDCL  = 26;  EINTR  = 27;
  ENTSK  = 28;  ENTTY  = 29;  EPIPE  = 30;  ELOCK  = 31;
type System_Error = ENONE..ELOCK;

 (* Define file access permissions *)

const
  FXSET = 6;
  FACOE = 5;  FACOW = 4;  FACOR = 3;
  FACUE = 2;  FACUW = 1;  FACUR = 0;
type perm_set = set of FACUR..FXSET;

 (* Define the file access types *)

const
  Read = 0;  Write = 1;  Execute = 2;  Read_Write = 2;
type  acc_set = set of Read..Execute;

 (* Define file types : Block, Character and Directory *)

const
  FSBLK = 1;   FSCHR = 2;   FSDIR = 3;
type file_modes = set of FSBLK..FSDIR;

 (* Define the tty flags for ttyget and ttyset system calls *)

const
  RAW   = 0;  ECHO  = 1;  XTABS = 2;
  LCASE = 3;  CRMOD = 4;  BSECH = 5;
  SCHR  = 6;  CNTRL = 7;
type flag_set = set of RAW..CNTRL;

type
  Time = record
           day,
           hour,
           minute,
           second,
           tick: integer
         end;

 (* A four-byte integer *)

  long_int = array [0..1] of integer;

 (* status record used in ofstat system call *)

  st_rec = record
             dev: integer; "device number"
             fdn: integer; "fdn number"
             mds: file_modes;
```

```
                prm: perm_set;
                cnt: integer;  "link count"
                own: integer;  "owner id"
                siz: long_int;  "file size in bytes"
                mtm: Time
              end;

  (* Time record for time system call *)

   tm_rec = record
              tim: Time;  "time of day"
              zon: integer;  "time zone"
              dst: boolean "daylight savings time flag"
            end;

  (* Task time information record used in ttime system call *)

   tt_rec = record
              usr,  "Task's user time"
              sys, "Task's system time"
              chu, "Children's user time"
              chs: Time  "Children's system time"
            end;

  (* tty record used in ttyget and ttyset system calls *)

   tty_rec = record
               flg: flag_set;  "Operation flags"
               dly: integer;  "Character delay "byte""
               prs: boolean;  "True is input characters available"
               cnc, bks: char  "Line cancel and Backspace chars"
             end;

   (* Standard buffer size used *)

const pagesize = 511;
type Page = array [0..pagesize] of char;

const  Default = 0;  Ignore = 1;  "for setting program interrupts"

 (* Heap manipulation routines *)

procedure mark(var TOP: integer);
procedure release(TOP: integer);

 (* Call another Pascal program *)

procedure run(ID: identifier; var PARAM: ARGLIST;
              var LINE: integer; var RESULT: PROGRESULT);

procedure abort;  "TERMINATE PROGRAM IMMEDIATELY"

function odd(X: integer): boolean;
```

```
(*****************************************************************)
{                    FILE I/O DEFINITIONS                         }
(*****************************************************************)

 (* Routines that begin with "_" or "__" may no be directly
    called by the programmer!  *)

procedure __GET(var F: text);
procedure __PUT(var F: text);

procedure __RDX(var C: char);
procedure __WRX(C: char);

function eoln(var F: univ text): boolean;
function eof(var F: univ text): boolean;

procedure __RLN;
procedure __WLN;

procedure __RWF(var F: text;  DUMMY1, DUMMY2: integer);
procedure __RWFS(var F: text);  " SHORT FORM of RWF "
procedure __EIO;
procedure __SETMSG;

procedure __RDI(var I: integer; WIDTH, DIGITS: integer);
procedure __RDC(var C: char; WIDTH, DIGITS: integer);

procedure __WRI(I: integer; WIDTH, DUMMY: integer);
procedure __WRC(C: char; WIDTH, DUMMY: integer);
procedure __WRS(S: identifier; WIDTH, DUMMY: integer);

procedure _FRESET(SIZE: integer; NAME: identifier);
procedure _FREWRITE(SIZE: integer; NAME: identifier);

procedure _SRESET;
procedure _SREWRITE;

procedure buffer(var F: text);  " TURN BUFFERING ON "
procedure unbuffer(var F: text);  " TURN BUFFERING OFF "

procedure page(var F: text);  " OUTPUT FORM FEED "

(*********************************************************)
{  Functions unique to the UniFlex programming environment  }
(*********************************************************)

procedure flush(var f: text);  "Flushes buffered output to 'f'"

procedure kill(which: identifier);  "Deletes a file"

" seek to an absolute position in a file.
 Used by the seek procedure "

procedure __seek(var f: text;  pos: integer; dir:direction);
```

```
 (* Execute programs in the UniFLEX Operating System *)

procedure system(command: univ identifier);

(*************************************************************)
{  Functions which map directly (more or less) onto UniFLEX  }
{  system calls.  Consult the UniFLEX Manual for details.    }
(*************************************************************)

procedure chacc(fname: identifier;  perm: acc_set;
   var error: System_Error);

procedure chdir(dirname: identifier;  var error: System_Error);

procedure chown(name: identifier;  owner: integer;
    var error: System_Error);

procedure chprm(fname: identifier;  perms: perm_set;
   var error: System_Error);

procedure cpint(which: integer;  kind: integer;
    var previous: integer;  var error: System_Error);

procedure create(fname: identifier;  perms: perm_set;
   var error: System_Error);

procedure crtsd(name: identifier;  desc: file_modes;
    perms: perm_set;  ddress: integer;  var error: System_Error);

procedure defacc(perms: perm_set);

procedure gtid(var task_id: integer);

procedure guid(var actual_id, effective_id: integer);

procedure link(fname1, fname2: identifier;  var error: System_Error);

 (* ofstat is like the system call "ofstat", but a file is passed
    instead of the file descriptor (fd) *)

procedure ofstat(var f: univ text;  var buf: st_rec;
    var error: System_Error);

 (* f_ofstat is like the UniFLEX system call "ofstat"; please
    consult the UniFLEX Manual for details.  *)

procedure f_ofstat(fd: integer;  var buf: st_rec;
    var error: System_Error);

procedure setpr(priority: integer);

procedure sleep(how_long: integer);
```

```
procedure status(name: identifier;  var buf: st_rec;
  var error: System_Error);

procedure suid(user_id: integer;  var error: System_Error);

procedure terminate(status: integer);

procedure time(var tm_buf: tm_rec);

procedure ttime(var tt_buf: tt_rec);

 (* Procedures ttyget and ttyset are similar to the system calls
    ttyget and ttyset; however, the Pascal procedures pass the
    actual file, not the file descriptor (fd).  Procedures f_ttyget
    and f_ttyset are just like the system calls ttyget and ttyset;
    these procedures do pass the file descriptor (fd).  *)

procedure ttyget(var f: univ text;  var tt_buf: tty_rec;
    var error: System_Error);

procedure f_ttyget(fd: integer;  var tt_buf: tty_rec;
    var error: System_Error);

procedure ttyset(var f: univ text;  tt_buf: tty_rec;
    var error: System_Error);

procedure f_ttyset(fd: integer;  tt_buf: tty_rec;
    var error: System_Error);

procedure unlink(fname: identifier;  var error: System_Error);

procedure update;

 (* Procedures fopen, fclose, fread, fwrite and fseek are similar to
    the system calls open, close, read, write and seek found in the
    UniFLEX Manual.  *)

procedure fopen(var fd: integer;  name: identifier; mode: integer;
  var error: System_Error);

procedure fclose(fd: integer;  var error: System_Error);

procedure fread(fd: integer; var buf: char;  size: integer;
  var count: integer;  var error: System_Error);

procedure fwrite(fd: integer;  var buf: char;  size: integer;
  var error: System_Error);

procedure fseek(fd: integer; position: long_int;
  kind: seek_kind;  var new_position: long_int;
  var error: System_Error);
```

```
 (* Conversion routines *)

function ctoi(var c: char): integer; "maps pairs of chars onto integers"

function btoi(var c: char): integer;  "maps a byte onto signed integers"

 (* Mathematical operations on "long" (four-byte) integers *)

procedure long_add(x, y: long_int;  var z: long_int);
procedure long_sub(x, y: long_int;  var z: long_int);
procedure long_mpy(x, y: long_int;  var z: long_int);
procedure long_div(x: long_int;  y: integer;
  var z: long_int;  var rem: integer);

procedure lock(var f: univ text;  n: integer;  var error: boolean);
procedure unlock(var f: univ text);

program P(var input, output: text;  var param: ARGLIST);
```

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.p 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:

```
++ xref Pascal-file-name [+l]
```

The Pascal-file-name is the UniFLEX 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.p
graph2.p
complex.p
setop.p
primes.p
minmax3.p
traversl.p
expon2.p
recurgcd.p
xref.p
```