COMPUTATION RESEARCH GROUP

CGTM NO. 196
NOVEMBER 1979

# STANFORD PASCAL COMPILER

SASSAN HAZEGHI

COMPUTATION RESEARCH GROUP
STANFORD LINEAR ACCELERATOR CENTER
STANFORD, CALIFORNIA 94305

[illegible]

[illegible]

## 1. Introduction

This note is a description of the Stanford Pascal system. It is also intended to serve as a guide for its setup, use and maintenance. The system consists of a machine independent Pascal compiler, a post-processor for adapting the output of the compiler to the IBM-370 system, a set of run-time support and interface routines and finally, a library of utility programs to help users in writing, debugging and evaluating Pascal programs.

Though primarily concerned with the IBM-370 implementation of the system, this writeup can also be of use to those interested in bootstraping the system onto other environments. Additional information about the bootstrap process and/or implementations on other machines can be found in references [4] through [7].

The rest of this writeup is organized as follows: Section 2 describes the compiler/post-processor and non-standard features of the Pascal implementation. Section 3 provides instructions and JCL for setting up the system. Section 4 explains some of the implementation details and provides a few storage saving ideas. Section 5 contains a sample program and demonstrates what kind of output to expect under various conditions. Section 6 indicates which features have changed from earlier versions of the Stanford Pascal system. Users of previous versions should take special note of this section. Sections 2.2.1, 2.2.3-2.2.9, 2.3.1, 2.4.1, 2.4.3, 3, 4.1 and 5 contain information of interest to the average users. Other parts of this note are meant primarily for the people who maintain the system or would like to modify it for their particular need. Access to reference [1] is also essential for all users.

## 2. Stanford Pascal Compiler

The Stanford Pascal Compiler is a modified version of the Zurich Pascal_P2 compiler (MAY 1974 variant) and except for a few minor extensions processes the same language (see [2] for more details on Pascal_P). The compiler itself is a 5000 line Pascal program that translates the source program into an intermediate form which is the machine language for a mythical Stack Computer (the so called P machine, hence the name P_Compiler). The output of the compiler is then fed to a post processor, the P_Translator, which in turn translates the P_Code into the IBM/370 code, generating either an Object Module or a 370 Assembly language program. The P_Translator is also written in Pascal (aprox. 4000 source lines) and like the compiler would benefit from any improvement in the code generation/translation of the combined system.

Except for a few cases involving the movement or comparison of large structures (i.e. large records, arrays, etc. implemented by the "Long" 370 "SS" type instructions), the translator generates instructions common to 370 and 360 series and, with small changes, it is possible to (optionally) generate 360-only instructions.

The translation from P_Code to 370 code is based on a general scheme for converting Polish style expressions into "Register" oriented code without actually simulating the Stack Machine on the Stack-less Computer which, due to lack of the hardware Stack and appropriate instructions, tends to be fairly inefficient. Furthermore, the organization of the translator is such that its modification to generate object code for other register oriented computers should be straightforward.

The run-time support package and the I/O interface is written to operate under OS/VS or OS/MVT and has also been tried by other users under VM. Using small I/O buffers (i.e. 10..12K bytes), the current version of the Compiler/Post_Processor can compile itself, and/or other moderate size programs, in a 128K region. A larger region, however, would improve the I/O efficiency.

### 2.1 The Sub_Monitor and I/O Interface

The Sub Monitor and the I/O interface consist of a set of assembly language routines which set up the run time environment and implement the I/O related Standard Procedures/Functions of Pascal. The sub monitor also initializes the environment for FORTRAN routines (by calling #IBCOM) if there are any FORTRAN routines present. They will be present if there are any explicit references to external FORTRAN routines in the Pascal program (see the section on external routines) or if any of the mathematical functions, SIN, COS, ARCTAN, LN, EXP or SQRT, are used. All references to these mathematical functions are implemented as calls to the corresponding FORTRAN double-precision functions.

## 2.2 Implementation Restrictions/Extensions etc.

The modifications to the Zurich Compiler are primarily in the areas of 1) providing TYPE information for certain instructions at the P_Code level, 2) boundary alignment of variables according to the 360/370 requirements and 3) separating CHARacters from INTEGERs in their internal representation. These changes should be transparent to the end user. Otherwise for a complete list of restrictions imposed by the P_Compiler refer to [2]. In addition:

### 2.2.1 Miscellaneous Restrictions

-Only TEXT files (FILE OF CHAR) are presently supported. For a method of circumventing this restriction, see Section 2.2.5.

-Files can be declared only in the main program (i.e. as global variables).

-Files can be passed only as VAR parameters to procedures or functions.

-Integers are limited to the range -2**31 to 2**31-1. This upper limit is the value specified by the constant MAXINT.

-Reals are implemented in the double-precision format on the IBM 360-370. This implies a precision of approx. 16 significant digits and a range of 10**-78 to 10**76 for the magnitude.

-Sets are limited to 64 elements. The ordinal range for the base type of the set must not extend outside the range 0..63.

-String constants are limited to a maximum length of 64 characters.

-Reals can be printed to only 12 digit accuracy (even though all real arithmetic is performed to 16 digit accuracy).

-A GOTO statement leading to a Label outside the procedure containing that statement is not allowed.

-The PACKED attribute in array and record declarations has no effect. All character/boolean arrays are always packed automatically with one element per byte. Standard procedures PACK and UNPACK, however, are supported and can operate on PACKED as well as unPACKED arrays.

-The standard procedure DISPOSE (as described in [1]) is not supported, instead, dynamic storage (acquired through the use of the standard procedure NEW) should be managed through the use of the predefined procedures MARK(P: Any_pointer_type); and RELEASE(P: Any_pointer_type). MARK is used to save the current value of the Heap pointer and RELEASE will reset the Heap pointer to the value specified by its (pointer type) argument. As an example, the following sequence:

```
    ... MARK(hp); ... NEW(x); ... NEW(y); ... RELEASE(hp); ...
```

leaves (the size of) the dynamic area unchanged. Note that the pointers "x" and "y" become "undefined" after the RELEASE operation and cannot be used before they are redefined. (Heap is the area from which dynamic storage is allocated.)

## 2.2.2 Storage Allocation for Variables

The Compiler allocates and aligns Pascal simple data types according to the following table :

| TYPE | SIZE | ALIGNED ON |
|---|---|---|
| CHAR,BOOLEAN | 1-BYTE | 1-BYTE BOUNDARY |
| INTEGER | 4-BYTES | 4-BYTE BOUNDARY |
| SET | 8-BYTES | 4-BYTE BOUNDARY |
| REAL | 8-BYTES | 8-BYTE BOUNDARY |

Dynamic storage, however, is always allocated on 8-Byte boundaries to avoid the necessity of alignment at run-time (as opposed to the Compile Time alignment). Note that Subranges are represented by their Base type and Enumerated types are treated as integers. The P+ compilation option (see 2.2.8) may be used to reduce the storage allocated to small integers, but at a cost in execution time. For the sake of space and time efficiency, it is a good practice to declare program variables in the order of their relative size. In particular, by defining simple type varibles before arrays and large records, you can ensure that all of the small variables may be accessed by a short address field, resulting in a shorter and somewhat faster program.

## 2.2.3 The Character Set and Pascal Identifiers

-Characters are internally represented by their EBCDIC value (i.e. ORD('a') = 129 = HEX'81'). Although this should be of no consequence to "clean" programs that make no assumption about the ordinal values of characters, one should note that: SUCC('a') = 'b'; but SUCC('i') <> 'j'. Furthermore, because of the size of the EBCDIC character set, the construct: SET OF CHAR; is not a valid type (use the July 77 version of the compiler if you have a pressing need for this feature).

-Identifiers may be of any length but only the first 12 characters are significant.

-Pascal keywords and other identifiers may contain upper and lower case letters interchangeably. For example, Ident and IDENT are treated as the same identifier.

-Identifiers may include the dollar and underscore ("$" and "_") characters wherever a digit may appear.

-The following symbols are treated identically by the input scanner of the compiler :

| | | | | | |
|---|---|---|---|---|---|
| '{' | or | '(*' | '@' | or | '¢' |
| '}' | or | '*)' | 'AND' | or | '&' |
| '[' | or | '(/' | 'OR' | or | '\|' |
| ']' | or | '/)' | 'NOT' | or | '¬' |

Note that comment brackets should be used consistently and a comment opened by the '(*' bracket cannot be closed with the '}' symbol.

-The Pascal 'uparrow' character is represented by '@' (the 'at sign' character).

-The '#' character (pound sign) is treated as a skip character and ignored by the compiler.

-The double-quote (") is used as a directive to skip text. All text up to and including the next double-quote is totally ignored.

-The above mentioned conversions do not apply to string constants in which the input characters are not subject to any automatic translation and/or interpretation.

## 2.2.4 Language Extensions

-The range designator A..B may be used to specify constant values A, A+1, ... B-1, B, instead of enumeration of all the individual values (e.g. [1, 4..8, 10, 12..20] is a good set constructor).

Note that this abbreviation may also be used in CASE labels. For example one can write:

```
        case CH of
           'a'..'i', 'A'..'I' : S1 ;
           '0'..'9'           : S2 ;
        end ;
```

-Superfluous separators ' ; ' preceding the END symbol in Record (and variant) declaration, Case Statement and Procedure definitions are ignored by the compiler.

-Functions of Type SET may be defined.

-The Tag field of a case variant record may be unnamed, in which case no space will be allocated for it. This feature, which is in the 'standard' language, allows access to different variants of a record when the type of each variant is known through some other context.

```
    e.g.,  record A: Some_type ;
           case BOOLEAN of
                TRUE:  ( B: Type_b) ;
                FALSE: ( C: Type_c) ;
           end ;
```

## 2.2.5 Files and File Handling

-The compiler khows about 6 predefined TEXT files, INPUT, OUTPUT, PRD, PRR, QRD and QRR, with INPUT used as input only, OUTPUT used as output only and PRD, PRR, QRD, QRR used as input after a RESET and as output after a REWRITE operation.

-The PROGRAM heading should include the names of all the predefined files used in the program, otherwise one has to RESET/REWRITE these (as well as all other user defined) files before they are accessed. Note that the default mode of INPUT, PRD and QRD is 'input' while OUTPUT, PRR and QRR are opened for 'output' if they appear in the PROGRAM parameter list. In order to use a predefined file in other than its default mode (e.g. to use PRD for 'output'), instead of listing it in the program heading, simply 'RESET' or 'REWRITE' that file prior to the relevent I/O operation(s).

-If the file name is missing from the argument list of a file-handling procedure/function then the file name INPUT or OUTPUT is inserted as appropriate. For example, READLN, READLN() and READLN(INPUT) are all equivalent as are PAGE, PAGE() and PAGE(OUTPUT).

-Boolean variables may be input from textfiles. The single letter 'T' represents TRUE and 'F' represents FALSE. Leading blanks are ignored and any other input character results in error. The file pointer is positioned to the charcter immediately following the 'T' or 'F' character. Note that lower-case input ('t' and 'f') is also accepted.

-String variables may be input using READ or READLN. For example, if S is a variable with the type ARRAY[1..N] OF CHAR, then READ(INPUT,S) is equivalent to:

```
for I:= 1 to N do
   if not EOLN(INPUT) then READ(INPUT,S[I])
      else S[I] := ' ';
```

-Only textfiles (file of CHAR) are currently supported. However the effect of other file types can be obtained through overlay techniques. For example, to use the PRR file as though it were declared as FILE OF REAL, the following code can be used:

```
var PRR_ELEMENT: record
                 case BOOLEAN of
                      TRUE:  (R: REAL);
                      FALSE: (CH: array[1..8] of CHAR)
                 end;

{ we omit other declarations, etc. }

PRR_ELEMENT.R := 0.5;  {Assign REAL value}
WRITE(PRR,PRR_ELEMENT.CH);  {Write it as a string}

{ input from the file can be performed similarly }
```

## 2.2.6 Additional Standard Procedures and Functions

CARD(S: Any_set_type) : returns an INTEGER result equal to the cardinality of the set, S. For example, CARD( [3,8,43,60] ) has the value 4.

CLOCK(I: INTEGER) returns an integer result corresponding to the value of the system clock. If I=0, the result is the execution time in thousandths of a second that have been used since the Pascal program started. Other values of I currently yield undefined results.

EXIT(I: INTEGER) : causes the Pascal program to terminate execution. The value, I, is used as the program's user return code and can be tested in the JCL used to run the program. The value used should be non-negative and less than 1000 to avoid confusion with the return codes used for Pascal errors.

EXPO(R: REAL) : returns an INTEGER result equal to the exponent in the internal machine representation of the real number, R. To use EXPO, it is necessary to know that real numbers are normalized in the form:

$$\pm \text{mantissa} * 16^{\pm \text{exponent}}$$

where 1/16 <= mantissa < 1 (except if the real number is zero then the mantissa is zero and the exponent is -64). For example, EXPO(1.0) is 1, EXPO(16.0) is 2, EXPO(256.0) is 3, etc. The EXPO function is useful for making fast determinations of the magnitude of a number (much faster than using the LN function).

LINELIMIT(F: TEXT; I: INTEGER) : sets a limit of I subsequent output lines for the file F. After I more lines have been written, an error message would be issued automatically. Initially, there are no limits in effect for any file. Performing a REWRITE or calling LINELIMIT with I<=0 will cancel any limit in effect for the file. If the file name, F, is omitted, OUTPUT is assumed.

MESSAGE(S: Any_string_type) : causes a character string to be written to the O.S. message log that is printed along with the JCL listing for the job. There is a limit of 120 characters on the length of this message.

MARK(P: Any_pointer_type) : saves the current value of the Heap pointer in the pointer variable P.

RELEASE(P: Any_pointer_type) : resets the heap pointer using the value of P. This effectively releases all the dynamic storage allocated (through the use of NEW) since the last MARK operation on P.

SKIP(F: TEXT; I: INTEGER) : if F is open for output, the effect is similar to I successive calls to WRITELN(F). When I=0, the next output line will overprint the current line. If F is open for input, the effect is similar to I successive calls to READLN(F). When I=0, the current input line will be re-read. If the file name, F, is omitted, OUTPUT is assumed.

SNAPSHOT(I,J: INTEGER) : causes a snapshot dump of active storage. This procedure requires access to the symbol table output of the compiler which is available only if the program is compiled with the D+ option in effect (see 2.2.8). The first parameter I specifies the number of active procedures/functions whose variables are to be printed. For example, I=3 specifies the 3 most recently entered procedures/functions. Specifying I=0 gives a dump of all active procedures/functions back to the main program. The second parameter J determines the type of dump. J=10 specifies the maximum amount of information is to be printed. J=0 is similar except that arrays are compressed by printing only the contents of the first few and last few elements. J=1 produces only a list of the active procedures and functions.

PACK/UNPACK : The restrictions on the type of the parameters to these standard procedures are somewhat relaxed. The target/source opernads need not be declard as PACKED arrays and, in addition, the source operand may be a string constant.

TRAP(I: INTEGER; VAR V: any type) : generates a call to an external user supplied routine with the entry point '$PASTRAP'. The value of I is passed in GPR-0 and the address of V is passed in GPR-1. The first parameter, I, is intended to be used as a 'function' code and the second parameter V is to pass values to/from the external routine. The object code for the external routine containing '$PASTRAP' entry point should be included with the object code of the Pascal program.

## 2.2.7 Predefined Names

-ALFA is defined to be the type ARRAY[1..10] OF CHAR.

-TEXT is defined to be the type FILE OF CHAR.

-MAXINT is defined to be an integer constant with the value 2147483647 (i.e. 2**31-1, the largest one-word integer value in the 360/370 series).

-DATE is a variable of type ALFA (ARRAY[1..10] OF CHAR) whose value is the date on which execution commenced. For example, '07-31-1979' corresponds to July 31, 1979.

-TIME is a variable of type ALFA that contains the time at which execution commenced. For example, '14:25:59 ' corresponds to 25 minutes and 59 seconds past 2 p.m.

-OSPARM is a pointer variable of type:

```
@RECORD
 LENGTH: INTEGER;
 STRING: ARRAY[1..64]
 END;
```

A parameter string may be passed to the Pascal program via the 'PARM' field of the 'EXEC' JCL staement (see Section 2.3.1 ). When this parameter string is supplied, OSPARM@.LENGTH is the number of characters in the string and the string itself is held in OSPARM@.STRING. When no parameter is provided, OSPARM has the value NIL. Note that the subscript bound of 64 is purely nominal and no attempt should be made to access elements of STRING with index values greater than the LENGTH value.

### 2.2.8 Compilation Options

Compiler Options are (as usual) specified inside COMMENT delimiters in any order, but with no other symbols/blanks between them. These options and their default values are :

```
(*$L+,M-,D+,K-,N-,X-,P-,C+,A-,S+,F+,E ... other comments*)
```

where:

| | |
|---|---|
| L+ | list/(don't list) source program. |
| M- | no margin/(set margin) at column 72 of input lines. |
| D+ | enable/(disable) run-time checking. |
| K- | don't emit/(emit) counters for program Run Profile. |
| N- | Do not nest/(allow nested) comments. |
| X- | clear/(set) external linkage flag. |
| P- | Do not pack/(do pack) subrange variables into "bytes". |
| C+ | emit/(don't emit) P_Code. |
| A- | gen. 370 Obj. Mod./(gen. 370 assembly language output). |
| S+ | save/(don't save) GPRs on procedure/function entry. |
| F+ | save/(don't save) FPRs on procedure/function entry. |
| E | Do a Page Eject before continuing the source listing. |

-M Option: The M option controls the margins for source input. When M- is in effect (the default), there are no margins and the entire input record is read by the compiler. When M+ is specified, a right margin at column 72 is set, so that columns 73 and beyond are ignored. M+ is useful for sequence numbered card input. More control over the margins of the input lines is provided by giving the M option in the form M(a,b). The first decimal number, a, specifies the left margin and b sets the right margin. That is, only the contents of columns a through b (inclusive) are compiled and the rest of the input line is ignored. No error occurs if b is given a value greater than the size of source records, the input lines are read to their ends in such a case. However there is a compiler limitation which restricts the maximum value of b to 120. (This limit may be changed by recompiling the compiler with a different value for the constant BUFLEN.) Consequently, M+ is equivalent to M(1,72) and M- is equivalent to M(1,120). Note: the M option does not come into effect until the following source record. If this proves inconvenient, observe that the M option can be placed in the JCL parameter string.

-D Option: With the D+ option in effect, various run-time checks are performed.

Subranges (including the Enumeration type variables) are checked when being assigned to or passed as actual parameters to procedures. Indices are checked before the indexing operation and Pointers are checked when being assigned to and/or before their use as references to other objects. Also, variables used in construction of Sets (through the set constructor operator [...]) or being tested for Set membership are checked to be within range prior to these operations. If the value being checked for validity happens to be a constant the appropriate check is done at Compile (really post processing) time, otherwise for the sake of conserving space, Run-Time check routine(s) are called to perform the proper tests (as opposed to in-line checking which would be more time-efficient).

If this option is in effect during compilation of a procedure heading, then the prologue of that procedure checks for the availability of sufficient storage on the run-time stack before allocating space for the local variables of the procedure. Similarly, the growth (and shrinking) of the Heap is checked to ensure the consistency of the Run-Time Stack/Heap structure. In order to detect uninitialized variables as early as possible, the entire stack/heap area as well as individual procedure activation records, are cleared to a fixed pattern (Hex '81'). (This can potentially make a significant contribution to the program's running time.)

Pointer values are checked before they are assigned and before they are dereferenced. The value must refer to a location within the storage area allocated to the heap. Also, the special pointer value NIL is valid on assignment but clearly invalid for dereferencing.

In case a Run-Time error is detected, the offending value with its declared range as well as the Procedure, and the relative location within the procedure, in which the error was discovered will be printed. If the D+ option is in effect while compiling the procedure heading, the approximate line number corresponding to the error location will also be given. If any of the above checks is possible at compile time, then the error message will be generated by the post-processor and the execution of the program will not be attempted. As the run-time diagnostic messages are sent to OUTPUT file, this file should be included in the set of Program files (i.e., DD statement for OUTPUT should be present).

Depending on the type of the checking, one to three full word instructions may be added to the object code per checking site. This means that a procedure which translates into almost 8k bytes of code, may exceed this limit when the D+ option is chosen. In such cases this option should be invoked either selectively, for small segments of the procedure, or the procedure should be broken down into smaller routines for debugging purposes (another incentive to avoid large procedures!).

-K Option: This option will cause the compiler to allocate counters and generate instructions needed to produce an execution profile of the user program. After the (proper) termination of the user Program with the above switch on, the Sub Monitor will output the counter values onto the QRR file, which should not be used by the user program. The Execution Profile Generator will then read these counts, as well as the source program listing and an auxiliary file generated by the compiler, in order to produce a formatted listing which includes the execution count of each (executable) line of the source program. The Execution Profile Generator and the necessary JCL are included in the TESTLIB file on the distribution tape. The Compiler usually generates a minimal number of additional instructions when this option is invoked, but in some marginal cases these extra instructions may cause a procedure to exceed the 8k size limitation, in which case the user may disable the Counts for that procedure or divide the procedure into smaller segments.

-N Option: When the 'N+' option is in effect, comment brackets may be properly nested. For example, (* (* *) (* *) *) would be a valid comment form. When 'N-' (the default) is in effect, the comment would be closed at the first "*)" bracket. Nested comments are useful when it is desired to comment-out sections of a Pascal program. Note that the option switches can be set only by the first level (outer most) comments.

-X Option: The immediate effect of this option is to change the CSECT names generated by the post-processor for Pascal procedures and the main program. Normally, the CSECT name for a procedure/function is formed from the first few characters of its name followed by a unique integer and the main program has the CSECT name $MAINBLK. While the 'X+' option is in effect, the CSECT names are taken directly from the procedure/function name. (Only the first 8 characters of the name are significant if used to create a CSECT name. It is the user's responsibility to ensure that the CSECT names are all distinct.) The main program is renamed to #MAINBLK (so that it is not automatically invoked by the sub-monitor program). The 'X+' option facilitates the creation of external Pascal procedures or functions. See Section 2.2.9, below.

-P Option: The Pack option 'P+' may be invoked universally, if the program does not use Dynamic REAL type variables (or records/arrays with REAL components), or selectively around procedures which need large data areas (either directly, through recursion or dynamic allocation etc.) to reduce the program's data space requirement. With the default value of the switch, Dynamic storage is allocated on double-word boundaries, with a potential for memory fragmentation. Furthermore, when this switch is on, scalar type variables which are in the range 0..255 are internally treated as CHARs, with one byte allocated per variable. Note that, in terms of running time, this representation is slightly less efficient than the standard representation of Scalar/Subrange Types as full word integers.

A byte-packed subrange variable cannot be passed as a reference (VAR) parameter to a procedure where the corresponding formal parameter is declared to be an INTEGER, nor can it be included in the parameter list of a READ statement. This may cause the compiler to 'find' some errors in an otherwise well-formed program when the Pack option is selected.

The 'P+' option is incompatible with the 'D+' setting and should not be specified when the run-time check is enabled. Otherwise the values of the variables, as printed by the SNAPSHOT routine, may not be accurate.

-F and S Options: If you have complicated REAL expressions involving call(s) to REAL functions in your program, you should leave the 'F+' switch ON, otherwise the 'F-' option would be more efficient. Likewise, if you do not use complicated expressions involving INTEGER valued Functions (and you have many procedure calls in your program), you may get a faster running program by a 'S-' option for the higher models of 370 (in which the LM/STM instructions are much slower than L/ST instructions).

Notes:

Only options L,D,M,K and E are of interest to the average user who should not be concerned with (and confused by) the details of the other switches. Options F and S should be used with care and some understanding of the code generation pattern of the compiler.

The option list (excluding the comment delimiters and the '$' tag) may be passed to the compiler through the 'PARM' field of the JCL 'EXEC' statement. This mode is particularly useful for interactive environments and avoids the need for editing the source program file in order to set/reset some of the option switches.

### 2.2.9 External Procedures

- Creating an External Pascal procedure

The simplest approach is to forego the usage of global variables within the external procedure. This procedure can then be compiled as part of a program that contains no global declarations, that sets the X+ compilation option and that contains no main program code. A small example of this is shown in Section 3.7. The object code created for the external procedure can be concatenated to the object code for the calling program before being link-edited or loaded into memory. The calling program must contain a declaration for the external procedure. The declaration follows the same syntax rules as for ordinary procedure definitions except that the code body is omitted. The keyword EXTERNAL simply follows the procedure heading.

It is possible for the calling program and the external procedure to share variables in the global environment. To do this, it is necessary to compile the external procedure using the identical global declarations as for the calling program. Also note that the SNAPSHOT routine cannot print the values of variables internal to an external procedure unless the symbol table file created for the external procedure during its compilation is saved. It must then be concatenated to the symbol table file created for the calling program during its compilation.

Note: As there is absolutely no Type/count checking provided by the Loader, it is important to make sure that the definition of the separately compiled Pascal/FORTRAN programs be consistent (in the number and Type of parameters) with the declaration of the corresponding procedure/function headings in the program making the calls. In the case of a separately compiled Pascal program, it is also important for the two declarations to have identical static nest levels if there is a potential two-way link (i.e. repeated cross calls) between the modules involved.

-Calling an External FORTRAN Routine

The FORTRAN function/subroutine should be declared as an internal Pascal function/procedure but with the keyword FORTRAN replacing the body of the code. Note that all reference type parameters should be declared as Pascal VAR parameters and the basic types INTEGER, REAL, CHAR and BOOLEAN in Pascal correspond to FORTRAN's INTEGER*4, REAL*8, LOGICAL*1 and LOGICAL*1 respectively. For example, to invoke FORTRAN's GAMMA function, the following code could be used:

```
     function DGAMMA(X: REAL): REAL;  FORTRAN;
       .
       .
     RESULT := DGAMMA(1.0);
```

Note: the double-precision versions of the FORTRAN routines should be used for guaranteed compatibility. However, single-precision versions will usually work correctly. If a Pascal REAL value is passed to a FORTRAN REAL*4 variable, some low-order digits are lost. If a result is returned from a FORTRAN REAL*4 expression to a Pascal REAL variable some undefined low-order digits are generated (implying that it may be impossible to return an exact zero result in these circumstances).

The FORTRAN message file FT06F001 should be present if you try to run a program which will call a FORTRAN routine. This is regardless of any I/O activity of the FORTRAN routine, for which you may have to include other DD statements as well. As the FORTRAN initialization routine (#IBCOM) tries to open this file at the entry to the monitor, the absence of this statement will cause an early (hard to diagnose) ABEND.

-Calling an External Assembler Routine

One method is to call an assembler routine via the TRAP built-in function. The routine must be given an entry point name of $PASTRAP. One of the routine's parameters may have any type. Consequently, any amount of information can be communicated via an appropriate record type parameter.

A second method is to code the routine to use FORTRAN parameter passing conventions and to call this routine as though it were a FORTRAN routine. The only drawback is the limitation of parameter types to those that have equivalents in FORTRAN.

Finally, the routine can be called as though it were a Pascal external routine. To access parameters and to return a result, some knowledge of the Pascal run-time organization is required. The calling program creates an activation record that is accessed via register 13. This record contains the parameters and a location to receive the returned result. The record's layout is shown in Section 4. The parameters correspond to the first few local variables. Note that for VAR parameters, it is the address of the argument that is placed in the activation record.

## 2.3 The Run-Time Environment

Prior to entry to the user program, the Pascal Sub Monitor acquires all the remaining storage in the user program's region, and returns a small portion of this space to the operating system to be used for I/O buffers. The rest of the storage area is shared between the run-time STACK, where program (compile time) variables are allocated, and the HEAP, which is used for allocation of dynamic storage (created explicitly by the programmer through the Standard Procedure NEW). The HEAP is internally organized as another stack, which grows/shrinks in the opposite direction of the variable allocation STACK, and it is the programmer's responsibility to ensure that the two do not run into each other in the course of the program's execution. The only notable restrictions imposed by the run-time environment on the source program are: 1) a limit of 10 distinct levels of static nesting of procedures, an arbitrary limit which may be increased if needed, and 2) a limit of 8K bytes on the size of individual procedures/functions (approx. 400..500 source lines).

### 2.3.1 JCL Parameter String

The user may specify the size of the run-time STACK/HEAP, the size of the area to be used for I/O buffers by the Operating System, the maximum running time of the program, the number of run-time errors to be tolerated, and generation of a memory dump in the 'PARM' field of the JCL 'EXEC' statement as follows:

```
// EXEC USERPROG,PARM='USER PARMS /STACK=xxxK,IOBUF=yyyK,
                ERRLIM=n,TIME=zzzS,NOSNAP,NOSPIE,NOCC,DUMP'
```

'USER PARMS': Parameter string to be passed to the user program (if any). The user program can access this string through the OSPARM built-in variable (see section 2.2.7).

'xxxK': Size of the storage area (in K bytes) to be allocated for the run-time Stack and Heap. This value, if not specified, defaults to the size of the largest obtainable contiguous area of memory minus the size of the I/O buffer area.

'yyyK': Size of the storage area (in K bytes) that is returned to the system for use as I/O buffers etc. This value which is independent of the Stack size parameter (i.e. the xxx value), will be defaulted to '36K', if not specified by the user. The default value, depending on the BLKSIZE of the files used in the program, should be sufficient for 6/8 files.

'n': The number of (non fatal) run-time errors that should be tolerated before the user program is terminated. The default value for 'n' is '1' and the program will normally stop execution after the first run time error is detected.

'zzzS': Maximum (estimated) running time of the program in seconds. If this parameter is present, the program will be stopped after the specified time limit.

'NOSNAP': This suppresses the automatic call to SNAPSHOT that is usually made when the Pascal program terminates due to an error. The option is useful if the symbol table file is unavailable, if the SNAPSHOT dump would waste too much paper or if the P+ compilation option was used.

'NOSPIE': This suppresses the interception of 'OCn' type abends by the sub-monitor. The option would only be useful when debugging by means of OS core dumps and for particularly stubborn errors (or when the bug is not in the Pascal program but in another program to which Pascal is linked).

'NOCC': When this keyword is NOT present, the first character on each output line may be consumed for character control purposes.

If NOCC is specified, no control characters are assumed and they will be automatically inserted by the sub-monitor. The use of NOCC implies that the only methods of controlling output spacing are the PAGE and SKIP built-in procedures.

'DUMP': This switch will cause an OS style memory dump to be generated when the number of run-time errors equals the 'ERRLIM'.

If the user program is entered through the Loader, the above parameter list should be included in the 'PARM' list to the Loader, and separated from the Loader parms by the '/' delimiter.

## 2.3.2 Invoking Pascal from Assembler Programs

Any Pascal program that has been saved as a load module (see Section 3) may be invoked from an assembler program. A typical calling sequence could be:

```
                .
                LINK   EP=PASCAL,PARAM=(PARM1),VL=1
                .
                .
PARM1           DC     H'13',CL13'/TIME=10,DUMP'
                .
```

The parameter corresponds to the JCL parameter string described in Section 3.1. It is set up as a halfword, containing the count of characters, immediately followed by those characters. The sub-monitor imposes a maximum length of 256 characters.
In common with many IBM-supplied processors, it is possible to provide a second parameter to specify ddnames that override those used in the Pascal program. Only the predefined ddnames (INPUT, OUTPUT, PRD, PRR, QRD, QRR) can be overridden. The second parameter consists of a halfword integer followed by a character string containing the replacement ddnames. The halfword integer must equal the number of characters in the string and it must be a multiple of 8. For example, to replace INPUT with SYSIN, OUTPUT with SYSPRINT, PRR with SYSUT1 and to leave the other ddnames unchanged:

```
                .
                LINK   EP=PASCAL,PARAM=(PARM1,PARM2),VL=1
                .
                .
PARM1           DC     H'13',CL13'/TIME=10,DUMP'
PARM2           DC     H'32'              Length of following list
                DC     CL8'SYSIN'         replaces INPUT
                DC     CL8'SYSPRINT'      replaces OUTPUT
                DC     XL8'0'             defaults to PRD
                DC     CL8'SYSUT1'        replaces PRR
                .
```

As seen in the example, an entry of binary zeros indicates that the built-in ddname is to be used. The entries in the list must be in order corresponding to INPUT, OUTPUT, PRD, PRR, QRD, QRR.

Before control is returned to the calling program, the sub-monitor closes all files used by the Pascal program and releases all dynamically acquired storage.

## 2.4 Compiler Outputs and Messages

Unless explicitly suppressd by the 'L-' option selector, the compiler generates a listing of the source program as it is read in. This listing also includes sequence number, static procedure/function nest level and program/data location counter fields on each line.

The Level field is used to indicate the static level at which each procedure or function is defined with the main program being at level 1. This column can be used to determine the scope of identifiers in the program and also clearly marks the beginning and ending of functions or procedures.

While processing variable declarations in each procedure or function, the Program/Data Location Counter field indicates the amount of storage allocated for local variables thus far. The same field shows the number of (intermediate) instructions generated for each procedure or function when the body (code section) of these routines are being compiled. At the end of each procedure/function this value shows the total number of instructions emitted up to that point in the program and it can be used as an indication of the size of the program being compiled. Each intermediate (P_CODE) instruction approximately corresponds to one 'RX' type (i.e. 4-byte) 370 instruction.

### 2.4.1 Compilation Error Messages

When the compiler finds a syntax error, it will place a marker ('@') pointing to the token past the position where the error was actually detected (i.e. one should search to the left and above the pointer to find the cause of the error). Each error indicator is followed by an 'error number' that corresponds to the codes given in the Pascal User Manual and Report [1]. At the end of compilation, the meanings for each error code that occurred are printed out. Runaway comments (i.e. comments with bad or missing closing brackets) can be easily located by the frozen value of the 'P/D LC' field and improper BEGIN/END nesting or missing end of procedure/functions can be traced with the help of the value in the 'LVL' field.

Note: error codes 398 and 399 correspond to implementation restrictions.

### 2.4.2 Post-Processor Error Messages

The following error codes mostly indicate that an internal table in the post-processor has overflowed. In such cases, the easiest fix is to split the Pascal program into smaller procedures/functions and to nest the procedures more deeply. If it is necessary to create a new version of the post-processor with larger table sizes, the appropriate change is indicated after the error message text below.

253- Procedure too long (larger than 8K bytes).
    --> Divide (the procedure) and conquer.
254- Too many long (string) constants.
    --> Recompile the Post_Processor with a larger value for MXSTR.
256- Too many Procedures/Functions referenced in this Proc.
    --> Recompile the Post_Processor with a larger value for MXPRC.
259- Expression too complicated.
    --> Simplify the expression by rearranging and/or breaking.
263- Too many (Compiler generated) Labels in this Procedure.
    --> Recompile the Post_Processor with a larger value for MXLBL.
281- Too many Integer constants in this Procedure.
    --> Recompile the Post_Processor with a larger value for MXINT
282- Too many Double Word (REAL,SET) constants in this Procedure.
    --> Recompile the Post_Processor with a larger value for MXDBL.
300- Divide by Zero (result of constant propagation).
    --> Fix up the (constant) expression evaluating to Zero.
302- Index/subrange value out of range (constant propagation ?)
    --> Fix up the (constant) expression to be within range.
501- Array component too large (larger than 32K).
    --> Reduce the range of the last (rightmost) indecies of the array and/or reorder the dimensions of the array so that they are ordered from the largest (leftmost) to the smallest (rightmost).

The following errors normally indicate an inconsistency in the Compiler and/or the Post_Processor. For more detail about these (and similar) messages refer to the source of the program issuing the message.

601- Type conflict of operands in the P_Program.
602- Operand should be of type 'ADR'.
604- Illegal type for run-time checking.
605- Operand should be of type 'BOOL'.
606- Undefined P_Instruction code.
607- Undefined Standard Procedure name.
608- Displacement field (of address) out of range.

609- 'Small' Proc Larger than 4K.
     --> Recompile the Post_Processor with "SHRT_PROC = 300".
611- Bad INTEGER alignment.
612- Bad REAL alignment.
613- Bad REAL constant.
614- Inconsistent Procedure Table file "PRD".
     --> Fix the JCL and/or the 'QRR' output of the compiler.

The error messages, if any, are followed by the name and the line number of the Procedure in which they are detected. If the Statement/expression causing the error cannot be easily identified, you should recompile the program with the 'A+' Option, listing the output of the Post_Processor (or the Input to 370/Assembler). See Section 3 (d) for an example of how to do this. As the source program line numbers appear at regular intervals in this outputg as well as the source program listing, it should be easy to associate the error message with its source.

## 2.4.3 Run-Time Errors

After a run-time error has occurred, there is usually an error message printed (either by SNAPSHOT or by the sub-monitor). However, in some circumstances (e.g., if no OUTPUT file is provided) it is necessary to deduce the problem from the user return code that is normally printed with the various operating system messages for the job. The return code value should be interpreted according to the following table.

| Return Code: | Implies: |
|---|---|
| 1001 | INDEX VALUE OUT OF RANGE |
| 1002 | SUBRANGE VALUE OUT OF RANGE |
| 1003 | ACTUAL PARAMETER OUT OF RANGE |
| 1004 | SET MEMBER OUT OF RANGE |
| 1005 | POINTER VALUE INVALID |
| 1006 | STACK/HEAP COLLISION |
| 1007 | ILLEGAL INPUT/RESET OPERATION |
| 1008 | ILLEGAL OUTPUT/REWRITE OPERATION |
| 1009 | SYNCHRONOUS I/O ERROR |
| 1010 | PROGRAM EXCEEDED THE SPECIFIED RUNNING TIME |
| 1011 | INVALID FILE DEFINITION |
| 1012 | NOT ENOUGH SPACE AVAILABLE |
| 1013 | *UNDEFINED OR OBSOLETE SUBMONITOR OPERATION* (should not occur) |
| 1014 | LINELIMIT EXCEEDED FOR OUTPUT FILE |
| 1020 | ILLEGAL INPUT PAST END OF FILE |
| 1021 | BAD BOOLEAN ON INPUT |
| 1022 | BAD INTEGER ON INPUT |
| 1023 | BAD REAL ON INPUT |

| | |
|---|---|
| 200X | PROGRAM INTERRUPTION CODE 'X' |
| 3001 | EXTERNAL ERROR (e.g. BAD PARAMETER TO MATH ROUTINES LOG, SQRT,.... etc) |
| X1XX | UNABLE TO CALL ON 'SNAPSHOT' AFTER A RUN ERROR (this happens if there is not enough space or if SNAPSHOT was not included in the Load Module or if the NOSNAP parameter was specified in JCL) OTHER DIGITS OF THE RETURN CODE TO BE INTERPRETED AS ABOVE |

NOTE: Return codes 1007 or 1008 could imply a bad or non-existant DD Statement for the accessed file, wrong direction for the I/O operation or an attempt to access a file prior to a RESET/REWRITE operation. Code 1009 usually implies that the file has conflicting DCB attributes.

In general, error messages point to the (approximate) location of the error within the Pascal program. Note however that the predefined files appearing in the program heading are opened on entry to the Pascal program and any problems that arise will cause error messages that refer to the beginning of the 'main' program (and not to any statements using the files).

Appendix A contains a complete directory of the error codes and messages generated by the compiler and the run-time system.

## 3. System Set-up and Maintenance Procedures

To bring up the system follow these steps:

a) Transfer File 5 from tape to disk, creating a card format PDS.

b) Perform link-edits to create load modules for the compiler (Pascal), the P-Code assembler (ASMPCODE), the run-time sub-monitor (PASCMON) and the run profile generator (PASPROF).

c) Set up your JCL and run some sample programs.

d) You may also want to create a Catalogued Procedure to avoid the bulky JCL for standard compilations.

The following are JCL samples you may find Helpful in creating the Load Modules and running programs. Note that the JCL statements provided here are meant to be used as a guideline and they may need to be modified before you can run them at your installation.

a) Copy file 5 (PASLIB) from the distribution tape to a disk. You should substitute the volume-serial numbers of a scratch disk and a disk to hold the Pascal object library for the names, WORK01 and DISK99, respectively. Warning: the control cards for IEHMOVE have a very rigid format. Continuations are signalled by a non-blank character in column 72; the continued text must begin in column 16 of the next card.

### 3.1 Copying Object Files from the Distribution Tape

```
//          JOB
//COPY      EXEC PGM=IEHMOVE
//SYSPRINT DD SYSOUT=A
//SYSUT1    DD UNIT=DISK,DISP=OLD,VOL=SER=WORK01
//SOURCE    DD UNIT=T9-1600,DISP=(OLD,KEEP),VOL=SER=PASCAL
//TARGET    DD UNIT=DISK,VOL=SER=DISK99,DISP=OLD
//SYSIN     DD *
 COPY DSNAME=WYL.CG.PAS.PASLIB,                              C
               FROM=T9-1600=(PASCAL,5),FROMDD=SOURCE,        C
               TO=DISK=DISK99,RENAME=Pascal.PASLIB,CATLG
//
```

### 3.2 Generation of Load Modules

```
//          JOB
//LKED      EXEC PGM=IEWL,PARM='MAP,NCAL'
```

```
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD   DD UNIT=DISK,DSN=PASCAL.SYSLMOD,
//             SPACE=(TRK,(4,3,1),RLSE),DISP=(NEW,CATLG)
//SYSPRINT DD SYSOUT=A
//OBJECT    DD DSN=PASCAL.PASLIB,DISP=SHR
//SYSLIN    DD *
  INCLUDE OBJECT(PASMONO,PASOBJ)
  ENTRY $PASENT
  NAME PASCAL
  INCLUDE OBJECT(PASMONO,ASMPOBJ)
  ENTRY $PASENT
  NAME ASMPCODE
  INCLUDE OBJECT(PASMON,PASSNAP)
  LIBRARY ($MAINBLK)
  ENTRY $PASENT
  ALIAS $PASENT
  NAME PASCMON
  INCLUDE OBJECT(PASMONO,PASPROF)
  ENTRY $PASENT
  NAME PASPROF
//
```

## 3.3 Running a Pascal Program

The following set up can be used to compile, post_process and run a user program. Note that the source program is read from COMPILE.INPUT, its listing is sent to COMPILE.OUTPUT, the intermediate code is sent to COMPILE.PRR and Procedure/Symbol/Counter tables are sent to COMPILE.QRR. COMPILE.QRR enables the Compiler to print an error log at the end of the source listing in case it detects any syntax error. The option list for the compiler may be passed in the 'PARM' field of the 'EXEC' card for the COMPILE step.

If there are no compilation errors, the Post-Processor generates an object module (POSTPROC.PRR) which is linked to the run-time monitor PASCMON. The routine SNAPSHOT forms part of the PASCMON module. SNAPSHOT can be called directly by the Pascal program or it may be called automatically by the sub-monitor in case of a run error. In either case, SNAPSHOT will access the GO.QRD file to read symbol table information. The D+ compilation option must be enabled for this information to be available.

If the K+ option is chosen in the source program, the PASPROF module is automatically invoked at the end of the GO step in order to print a brief summary of the statement execution frequencies. If you wish a full execution profile listing, follow the instructions given in Section 3.6. The GO.PRD DD statement is required by the execution profiler.

Note: this JCL is set up in the form of an "in-stream" JCL procedure. Ideally, the JCL procedure should be copied into the catalogued JCL procedures library at your installation. The procedure consists of the PROC JCL card and the following cards, up to but not including the PEND card. Also observe that only the member PASCMON (with alias name $PASENT) is loaded from the call library PASCAL.PASLMOD listed in the GO.SYSLIB DD statement. It may be more appropriate to remove PASCMON from this file and place it in some other load module library that is referenced in GO.SYSLIB.

```
//          JOB
//PASCAL    PROC GOTIME=10
//*
//*    STEP ONE:  COMPILE THE SOURCE PROGRAM
//*
//COMPILE   EXEC PGM=PASCAL,COND=(0,LT)
//STEPLIB   DD DSN=PASCAL.PASLMOD,DISP=SHR
//OUTPUT    DD SYSOUT=A
//PRD       DD DSN=PASCAL.PASLIB(PASMSG),DISP=SHR
//PRR       DD DSN=&&PCODE,UNIT=SYSDA,DCB=RECFM=VB,
//             SPACE=(TRK,(20,5),RLSE),DISP=(,PASS)
//QRR       DD DSN=&&TABLES,UNIT=SYSDA,DCB=RECFM=VB,
//             SPACE=(TRK,(5,2),RLSE),DISP=(,PASS)
//*
//*    STEP TWO:  (POST) PROCESS THE P_CODE
//*
//POSTPROC  EXEC PGM=ASMPCODE,COND=(0,LT)
//STEPLIB   DD DSN=PASCAL.PASLMOD,DISP=SHR
//INPUT     DD DSN=*.COMPILE.PRR,DISP=(OLD,DELETE)
//PRD       DD DSN=*.COMPILE.QRR,DISP=(OLD,PASS)
//OUTPUT    DD SYSOUT=A
//PRR       DD DSN=&&OBJECT,UNIT=SYSDA,DCB=RECFM=FB,
//             SPACE=(TRK,(10,5),RLSE),DISP=(,PASS)
//*
//*     STEP THREE:  LOAD AND GO
//*
//GO        EXEC PGM=LOADER,COND=(0,LT),PARM='//TIME=&GOTIME'
//STEPLIB   DD DSN=PASCAL.PASLMOD,DISP=SHR   (NEEDED FOR K+ ONLY)
//SYSLIN    DD DSN=*.POSTPROC.PRR,DISP=(OLD,DELETE)
//SYSLOUT   DD SYSOUT=A
//SYSLIB    DD DSN=SYS1.FORTLIB,DISP=SHR
//          DD DSN=PASCAL.PASLMOD,DISP=SHR
//PRD       DD DUMMY
//QRD       DD DSN=*.COMPILE.QRR,DISP=(OLD,DELETE)
//QRR       DD UNIT=SYSDA,SPACE=(TRK,(2,2))
//FT06F001  DD SYSOUT=A
//OUTPUT    DD SYSOUT=A
//          PEND
//*
//RUN       EXEC PASCAL,PARM.COMPILE='compilation option list'
//COMPILE.INPUT DD *
```

```
    (* the Pascal source program *)

//GO.INPUT DD *
  (* input data, if any *)
//
```

## 3.4 Inspection of Generated Code

This procedure can be used to inspect the 370/Assemblly code generated by the compiler. The JCL assumes that the (JCL) procedure, PASCAL, has been catalogued. If it has not, you must insert the procedure definition as given in (b) after the JOB card. Note: the assembly code that is produced can be combined with the macro definitions that are provided in Pascal.PASLIB(PBGN) and then assembled, loaded and executed. However, this mode of operation is not recommended (the assembly time could be 3-4 times that of the compilation time).

```
//       JOB
//PEEK       EXEC PASCAL,PARM.COMPILE='A+',COND.GO=(0,LE)
//COMPILE.INPUT DD *

(* Pascal source program, including other options. NOTE 'A+' *)

//POSTPROC.PRR DD SYSOUT=A
//
```

## 3.5 Saving Pascal Programs as Load Modules

In order to create a Load Module from a Pascal program the following set up can be used. The JCL assumes that a dataset, ARTHUR.LOAD, has previously been allocated and catalogued and will be used to hold the created load module named BILL. As before, it assumes that PASCAL is a catalogued JCL procedure. If it is not, add the procedure definition given in (b) to the beginning of this card deck. If you are creating a new version of a compiler program (i.e., the PASCAL or ASMPCODE load modules) you should substitute PASCAL.PASLIB(PASMONO) for Pascal.PASLMOD(PASCMON) in the JCL. This is to use the smaller faster version of the sub-monitor that is recommended for "safe" programs. If you do not make this substitution, no harm will result.

```
//          JOB
//SAVE      EXEC PASCAL,COND.GO=(0,LE)
//COMPILE.INPUT DD *

  (* Pascal source program *)
```

```
//LKED      EXEC PGM=IEWL,PARM='MAP,LET'
//SYSUT1    DD UNIT=SYSDA,SPACE=(TRK,(50,50))
//SYSLMOD   DD DSN=ARTHUR.LOAD(BILL),DISP=OLD
//SYSPRINT  DD SYSOUT=A
//SYSLIB    DD DSN=SYS1.FORTLIB,DISP=SHR
//SYSLIN    DD DSN=PASCAL.PASLMOD(PASCMON),DISP=SHR
//          DD DSN=&&OBJECT,UNIT=SYSDA,DISP=(OLD,DELETE)
//          DD *
 ENTRY $PASENT
//
```

To run a Pascal program that has been saved as a load module, the following pattern of JCL may be used:

```
//          JOB
//GO        EXEC PGM=BILL,PARM='/NOSNAP,TIME=15'
//STEPLIB   DD DSN=ARTHUR.BILL,DISP=SHR
//OUTPUT    DD SYSOUT=A
//FT06F001  DD SYSOUT=A
//*
//*         DD CARDS FOR OTHER FILES, IF NEEDED.
//*
//INPUT     DD *
  (* Input Data - if required *)
//
```

## 3.6 Generation of Execution Profiles

The standard JCL setup (section 3.3) will allow a brief (very condensed) execution profile to be printed if the K+ compilation option is used. In order to generate the full profile (a program listing with execution frequencies alongside each statement), either of the following schemes can be used. The first method is quite simple but it requires the Pascal source program to be available in a disk file. Suppose it is stored and catalogued under the name SOURCE.PASCAL. The following job could then produce the desired profile.

```
//          JOB
//PROFILE   EXEC PASCAL,PARM.COMPILE='K+'
//COMPILE.INPUT DD DSN=SOURCE.PASCAL,DISP=SHR
//GO.PRD    DD DSN=SOURCE.PASCAL,DISP=SHR
//GO.INPUT DD *
  (* Input Data - if required *)
//
```

Alternatively, the following JCL can be used to generate a compiler-formatted program profile. Some of the extra JCL is to ensure that a source listing is produced even if the COMPILE or

POSTPROC steps terminate with error(s). As in 3.3, the JCL is arranged to use an in-stream procedure. This procedure, PASCALK, should ideally be added to your installation's catalogued procedures library also.

```
//          JOB
//PASCALK   PROC GOTIME=10
//*
//*    STEP ONE:  COMPILE THE SOURCE PROGRAM
//*
//COMPILE   EXEC PGM=PASCAL,PARM='K+',COND=(0,LT)
//STEPLIB   DD DSN=PASCAL.PASLMOD,DISP=SHR
//OUTPUT    DD DSN=&&LISTING,UNIT=SYSDA,
//             SPACE=(TRK,(10,10),RLSE),DISP=(,PASS)
//PRD       DD DSN=PASCAL.PASRLIB(PASMSG),DISP=SHR
//PRR       DD DSN=&&PCODE,UNIT=SYSDA,DCB=RECFM=VB,
//             SPACE=(TRK,(20,5),RLSE),DISP=(,PASS)
//QRR       DD DSN=&&TABLES,UNIT=SYSDA,DCB=RECFM=VB,
//             SPACE=(TRK,(5,5),RLSE),DISP=(,PASS)
//*
//*    STEP TWO:  (POST) PROCESS THE P_CODE
//*
//POSTPROC EXEC PGM=ASMPCODE,COND=(0,LT)
//STEPLIB   DD DSN=PASCAL.PASLMOD,DISP=SHR
//INPUT     DD DSN=*.COMPILE.PRR,DISP=(OLD,DELETE)
//PRD       DD DSN=*.COMPILE.QRR,DISP=(OLD,PASS)
//OUTPUT    DD SYSOUT=A
//PRR       DD DSN=&&OBJECT,UNIT=SYSDA,DCB=RECFM=FB,
//             SPACE=(TRK,(10,5),RLSE),DISP=(,PASS)
//*
//*     STEP THREE:  LOAD AND GO
//*
//GO        EXEC PGM=LOADER,COND=(0,LT),PARM='//TIME=&GOTIME'
//STEPLIB   DD DSN=PASCAL.PASLMOD,DISP=SHR
//SYSLIN    DD DSN=*.POSTPROC.PRR,DISP=(OLD,DELETE)
//SYSLIB    DD DSN=SYS1.FORTLIB,DISP=SHR
//          DD DSN=PASCAL.PASLMOD,DISP=SHR
//SYSLOUT   DD SYSOUT=A
//SYSTERM   DD SYSOUT=A
//PRD       DD DSN=*.COMPILE.OUTPUT,DISP=(OLD,PASS)
//QRD       DD DSN=*.COMPILE.QRR,DISP=(OLD,PASS)
//QRR       DD UNIT=SYSDA,SPACE=(TRK,(5,2))
//OUTPUT    DD SYSOUT=A
//FT06F001 DD SYSOUT=A
//*
//*     PRINT THE SOURCE PROGRAM IF ANY STEP FAILED
//*
//LISTSRC   EXEC PGM=IEBGENER,COND=((1000,GT,GO),EVEN)
//SYSPRINT DD DUMMY
//SYSIN     DD DUMMY
//SYSUT1    DD DSN=*.COMPILE.OUTPUT,DISP=(OLD,DELETE)
```

```
//SYSUT2   DD SYSOUT=A
//*
//         PEND
//*
//RUNCOUNT EXEC PASCALK
//COMPILE.INPUT DD *

  (* Pascal source program *)

//GO.INPUT DD *
  (* input data - if any *)
//
```

Note that the profiler's actions are controlled by the input that it receives in the QRR file. If the file is empty (as it is with the JCL is Section 3.3), a brief summary is generated. Otherwise it may contain the program text (either as the source form or in the compilation output form) in which case, a program listing with statement execution frequencies on the left is printed.

## 3.7 Using External Pascal Procedures

To compile and save an external Pascal procedure/function as an object module. The existance of a pre-allocated, catalogued dataset CHARLIE.OBJECT to hold the object code is assumed. Example coding:

```
//         JOB
//SAVE     EXEC PASCAL,PARM.COMPILE='X+',COND.GO=(0,LE)
//COMPILE.INPUT DD *
  (*  Note the X+ option   *)
  PROGRAM DUMMY;
     PROCEDURE EXTRT( PARM1, PARM2: REAL );
        VAR X,Y,Z: INTEGER;
        BEGIN
           (* Body of the external routine *)
        END;
     BEGIN
        (* No main program code *)
     END.
//POSTPROC.PRR DD DSN=CHARLIE.OBJECT,DISP=OLD
//
```

An example of how to use this saved external procedure now follows:

```
//         JOB
//RUN      EXEC PASCAL
```

```
//COMPILE.INPUT DD *
  PROGRAM MAIN(INPUT,OUTPUT);
     VAR S1, S2: REAL;
         (* other declarations omitted *)
     PROCEDURE EXTRT( PARM1, PARM2: REAL );  EXTERNAL;
         (* other procedure/function defs omitted *)
  BEGIN
     .
     EXTRT( S1, S2 );  (* invoke the routine *)
     .
  END.
//GO.SYSLIN DD
//           DD
//           DD DSN=CHARLIE.OBJECT,DISP=SHR
//GO.INPUT   DD *
   (* input data - if any *)
//
```

## 3.8 Printing the Documentation File

To print another copy of this document, the following job may be submitted. Note that the SYSUT2 output must be transmitted to a device that supports the full upper/lower-case character set.

```
//          JOB
//LIST      EXEC PGM=IEBPTPCH
//SYSPRINT  DD SYSOUT=A
//SYSUT1    DD UNIT=T9-1600,DISP=(OLD,KEEP),VOL=SER=PASCAL,
//             LABEL=(1,SL),DSN=WYL.CG.PAS.PASDOC
//SYSUT2    DD SYSOUT=A   UPPER/LOWER-CASE PRINTER
//SYSIN     DD *
  PRINT  PREFORM=A,MAXFLDS=1
  RECORD FIELD=(80)
//
```

## 4. Some Implementation Details

The Sub Monitor, entered via the $PASENT entry point, acquires all the space available to the user program, releasing some 36K bytes of it for I/O buffers, and sets up the run-time STACK/HEAP as well as the appropriate registers. It then calls the user program (at $MAINBLK) and eventually regains control upon proper termination of the program or a call to the EXIT routine within the program. The monitor, if returned to through a call to EXIT, will return the argument of the EXIT as the Step Return code, otherwise it will return a zero value.

### 4.1 I/O and File Structure

The I/O routines handle all the operations on the Predefined Files, with each file having its own set of flags and data control block. Locate-mode I/O is used universally and this implies that there is effectively no limit on the file record sizes other than the amount of storage available for system buffers (controlled by the IOBUF parameter). Most file formats are supported. The following list shows all the allowed combinations of RECFM attributes:

( F or V ) [ B ] [ S ] [ A or M ]  or  U

where square brackets enclose an optional choice and round brackets enclose a compulsory choice. For example, FBSA and VB are allowed combinations.

There is one minor quirk. Due to a basic incompatibility between locate-mode I/O and U-format records, all output U-format records are written with their maximum length. However, a file containing U-format records with varying sizes can be read correctly.

Output lines destined for F and U format files are padded with blanks at their right ends so as to achieve the required LRECL for the file. V format files do not require such padding and none is performed - except that the operating system will not accept completely empty lines, these are replaced with lines containing a single blank.

Over-long output lines (i.e., they contain more characters than the file's LRECL value) are split whenever the LRECL value is exceeded.

If DCB attributes are omitted from the JCL (and are not available from the dataset control block) the sub-monitor will supply reasonable defaults. The default values are chosen according to the following rules (the rules must be applied in the order given):

1- If RECFM is unspecified, it defaults to VB for all files except OUTPUT; for that file it defaults to VBA.

2- If LRECL is unspecified, it defaults to a basic value of 80 for all files except OUTPUT; for that file it is 132. If the RECFM includes the V attribute, 4 is added to the basic value. If the RECFM includes the A or M attributes, an additional 1 is added.

3- If the BLKSIZE is unspecified, then the default depends on whether the RECFM includes the V, F or U attribute.
V: BLKSIZE is set to 1600 for all files except OUTPUT; for that file it is 3200.
F: BLKSIZE is chosen to be the largest multiple of LRECL that does not exceed the numbers given above for RECFM=V. However, if this would cause BLKSIZE to be zero, then BLKSIZE is made equal to the LRECL value.
U: The BLKSIZE is set equal to the LRECL.

4- If BUFNO is unspecified, it defaults to 3 for all files except OUTPUT; for that file it defaults to 5.

To conform to the specification of the Revised Report of Pascal, an extra blank is inserted at the end of every record of a textfile on input. For example, if F is a textfile then successive calls to GET(F) will step F@ through all the characters in the current input record. When F@ is the last character, another call to GET(F) will cause F@ to be a blank and EOLN(F) to become True. One more call of GET(F) will step F@ to the first character of the next record.

At the end of a textfile, the actions are as follows. Suppose that F@ refers to the last character in the last record of the input file F. Then a call to GET(F) will make EOLN(F) true and make F@ be a blank, however EOF(F) is still false. One more call of GET(F) causes EOLN(F) and EOF(F) to both be true and F@ is still a blank. More calls of GET(F) do not change this situation.

Character-by-character input beyond the end-of-file marker does not cause a run-time error - blanks are simply read. However, any attempt to read a Boolean, Integer or Real value past the end of file causes a run-time error.

## 4.2 Procedure/Function Call Mechanism and Stack Organization.

Procedure Calls follow the usual OS conventions. In addition, register 12 (GPR 12) points to the base (bottom) of the STACK, serving as the base register for the GLOBAL variables. GPR 13 points to the base of the data area (activation record) of the currently active procedure, serving as Base Register for the (very) LOCAL variables. Everything in between (i.e. non LOCAL, non GLOBAL) is accessed by loading the base address of the associated activation record from the DISPLAY table into a temporary register (GPR 14 or 1). The DISPLAY table, consisting of 1 entry per static nesting level of the program, is within the GLOBAL data frame and thus always accessible. Note that GLOBAL program variables start after the CHARacter File buffers and the variables defined within procedures, depending on whether the FPRs are saved or not, start after the FPR Save Area or Function result location. This scheme allows GPR 13 to point to a Register Save Area (with the usual forward/backward links) while being the LOCAL data Base Register at the same time.

The current value of the HEAP pointer is kept in the location following the GPR Save Area and this location corresponds to the 'NP' register of the P_Machine. GPR 10 and 11 are used as Base Registers for the currently active Procedure and GPR 2..9 as well as FPR 2..4 make up the expression evaluation stack. For more information on the organization of Run-Time stack and the use of the Display Table see [3] and [4].

The following table shows the state of the STACK/HEAP structure while running a Pascal program.

```
STACK

GPR12-->   000- GLOBAL  (bottom of run-time STACK)
           004- Back Link, Save Area.
           008- Forward Link, Save Area.
           012- GPR Save Area, (GPR14..GPR12).
            .
            .
           072- Current HEAP (NEW) Pointer, 'NP'.
           076- End of Heap Pointer, 'NPO'.
           080- FPR Save Area.
            .
            .
           112- Fix/Float Conversion Constants. (4 Double Words)
            .
            .
           144- DISPLAY[1]
            .     .
            .     .
           180- DISPLAY[10]
           248- INPUT@  (INPUT file buffer)
           249- OUTPUT@ (OUTPUT file buffer)
           250- PRD@    (PRD file buffer)
           251- PRR@    (PRD file buffer)
           252- QRD@    (QRD file Buffer)
           253- QRR@    (QRR file buffer)
            .           (buffers for other files)
            .
           280- DATE
```

```
                 290- TIME
                 300- OSPARM
                 304- First (user declared) GLOBAL program variable.
                   .
                   .
                   .
                   .

      GPR13-->   8n+0 LOCAL  (current Stack Frame)
                 +004 Back Link, Save Area.
                 +008 Forward Link, Save Area (NIL at this time).
                 +012 GPR Save Area, (GPR14..GPR12).
                   .
                   .
                 +072 FUNCTION result, (unused in case of PROCEDUREs)
                 +080 FPR save area (optional)
                 +080 LOCAL  (first local variable if FPRs not saved)
                   .
                   .
                 +112 LOCAL  (first local variable, if FPRs saved)
                   .
                   .
                   .
                   .
      NP -->     8m   Next (to be) allocated DYNAMIC variable.
                   .
                   .       (HEAP area already allocated)
                   .
      NPO -->    8j   End of HEAP and user data space.

      HEAP
```

Note: Program variables are allocated in the order of declaration within each declaration group and the address appearing in the source listing produced by the compiler, is the address of the first variable allocated in that group. For example the program listing:

```
   1 304 0   PROGRAM NONSENSE(OUTPUT) ;
   2 304 1   VAR  I, J, K : INTEGER;
   3 316 1        CH, NXTCH : CHAR ;
   4 318 1        ...
                  ...
```

```
means that Location 304 is assigned to the variable I,
                    308                             J,
                    312                             K,
                    316                             CH,
                    317                             NEXTCH,
etc.
```

The comments preceding the source code of the compiler, postprocessor and the I/O module also provide some useful information for those interested in the organization of the run-time environment.

## 4.3 Hints on Run Time Errors

In case you encounter a run-time error while running a program (i.e. a program ABEND), first check the following points before resorting to the OS generated DUMP.

1) See if the appropriate options are specified (e.g., you should not run a program with the C- option selected).

2) Make sure all the files used in the program appear in the parameter list of the PROGRAM statement, and/or they are RESETed/REWRITten before any operation takes place. Also note that the direction of operation should be compatible with the file and/or the previous RESET/REWRITE on that file (i.e. no READ from output or after a REWRITE etc.) If the run-time check is enabled (either by default or an explicit 'D+') or a Run Profile (execution frequency of program statements) is requested by the 'K+' switch, it is important that the JCL for the additional Symbol Table and/or Counter files are properly included in the user program. A missing or incorrect DD statement for such files may cause the program to be terminated in the Pascal monitor without a clear connection to the user program.

3) Check that there is a DD statement for every file used in the program and RECFM, LRECL and BLKSIZE have acceptable values.

4) The size of the region in which you run the program should be sufficient to accommodate the code as well as data. The program listing gives you an approximate idea of the size of the program and the data area. Recursive procedures however, depending on how deep the recursion goes, may need much more space than the size of their local variables may suggest. You can check to see if the run-time STACK and HEAP are colliding by comparing the HEAP pointer (at GPR12@+72) and GPR13 which points to the base of the LOCAL data area.

5) Check for bad (uninitialized, out of range) indices as well as illegal pointer references caused by uninitialized/NIL pointers in the procedure causing the ABEND.

Also see the extended run-time checking facilities (the D+ compilation option).

## 4.4 Storage Saving Considerations

In general the P_Code assembler trades memory for speed and, in particular, it prefers a sequence of RX and RR type 370 instructions over the corresponding SS type instructions which tend to be more compact though usually slower (the difference is quite noticeable on the larger 370 models). However, it is possible to reduce the storage requirement of your program in certain cases.

1) Dynamic storage is currently allocated on 8-byte boundaries. If you do not use this kind of storage for REAL values, you can change the alignment factor to 4 (= INTSIZE) as opposed to 8 (= REALSIZE) in the Procedure NEW1 of the Compiler. This should improve memory usage specially if dynamic storage is heavily used.

2) The current sub-monitor releases some 36K bytes of storage to be used for I/O buffers. This space could be reduced to as little as 8K, leaving the rest for the user program, by using smaller BLKSIZEs for the files. By reducing the above space to 8K, you can compile the Compiler in a 128K region.

3) If you group variables and fields with the same (internal) type together, you may improve the storage utilization by cutting down on fragmentation of the memory. This is particularly important in the case of ARRAYs OF RECORDs which contain fields of different types. The rearrangement of fields, however, should not be done at the expense of clarity and logical continuity of data declarations.

4) See the Pack Option in section 2.3.8.

## 5. Examples

The following program (a small deviation from the standard Factorial example) shows a simple -and very expensive- way of generating a table of Fibonacci Numbers and it is also meant to illustrate the Compilation, Post_Processing and Execution of a typical Pascal program. The compiler output has been slightly edited to compress its width across the page.

" Sample Program, including the necessary JCL "

```
//          JOB
//TEST      EXEC PASCAL
//COMPILE.INPUT DD *

PROGRAM fib_demo(OUTPUT) ;

TYPE pos_int = 0..30 ;

VAR  i     : pos_int ;
   time   : INTEGER ;

 FUNCTION fibonacci(j :pos_int) : INTEGER ;
 (*To evaluate fibonacci # j, for j >= 0,
                      subject to integer overflow*)

  BEGIN
  IF j = 0 THEN fibonacci := 0
  ELSE IF j = 1 THEN fibonacci := 1
    ELSE fibonacci := fibonacci(j-1) + fibonacci(j-2) ;
  END ;

 BEGIN (*fib_demo*)
 FOR i := 10 TO 25 DO
  BEGIN  time := CLOCK(0) ;
  WRITELN(' Fibonacci # ', i:3, ' is :', fibonacci(i):6,
  '  (Compute time =', CLOCK(0)-time:5, ' Milli Sec.)') ;
  END ;
 END.
//
```

" Source Program listing generated by the Compiler "

```
LINE # P/D LC LVL < Stanford Pascal Compiler, Version of July-78 >

   1    288  1) PROGRAM fib_demo(OUTPUT);
   2    288  1)
   3    288  1) TYPE pos_int = 0..30;
```

```
288 1)
288 1) VAR  i     : pos_int;
292 1)      time : INTEGER;
296 1)
296 1)  FUNCTION fibonacci(j :pos_int) : INTEGER;
 84 2)  (*To evaluate fibonacci # j, for j >= 0,
                            subject to integer overflow*)
 84 2)
 84 2)   BEGIN
  0 2)   IF j = 0 THEN fibonacci := 0
  5 2)   ELSE IF j = 1 THEN fibonacci := 1
 12 2)     ELSE fibonacci := fibonacci(j-1) + fibonacci(j-2);
 29 2)   END;
 84 2)
 84 2)  BEGIN (*fib_demo*)
  0 1)  FOR i := 10 TO 25 DO
 15 1)   BEGIN  time := CLOCK(0);
 18 1)   WRITELN(' Fibonacci # ', i:3, ' is :', fibonacci(i):6,
 37 1)   '  (Compute time =', CLOCK(0)-time:5, ' Milli Sec.)');
 53 1)   END;
 62 1)  END.
```

```
****   NO SYNTAX ERROR(S) DETECTED.

****   23 LINE(S) READ,  1 PROCEDURE(S) COMPILED,

****   94 P_INSTRUCTIONS GENERATED,  0.04 SECONDS IN COMPILATION.
```

" Post_Processor messages "

```
****   NO ASSEMBLY ERROR(S) DETECTED.

****   672 BYTES OF CODE GENERATED, 0.05 SECONDS IN P_CODE ASSEMBLY.
```

" Output of the Sample Program "

```
Fibonacci # 10 is :    55  (Compute time =    4 Milli Sec.)
Fibonacci # 11 is :    89  (Compute time =    5 Milli Sec.)
Fibonacci # 12 is :   144  (Compute time =    8 Milli Sec.)
Fibonacci # 13 is :   233  (Compute time =   12 Milli Sec.)
Fibonacci # 14 is :   377  (Compute time =   19 Milli Sec.)
Fibonacci # 15 is :   610  (Compute time =   31 Milli Sec.)
Fibonacci # 16 is :   987  (Compute time =   51 Milli Sec.)
Fibonacci # 17 is :  1597  (Compute time =   83 Milli Sec.)
Fibonacci # 18 is :  2584  (Compute time =  133 Milli Sec.)
Fibonacci # 19 is :  4181  (Compute time =  215 Milli Sec.)
Fibonacci # 20 is :  6765  (Compute time =  348 Milli Sec.)
Fibonacci # 21 is : 10946  (Compute time =  565 Milli Sec.)
Fibonacci # 22 is : 17711  (Compute time =  914 Milli Sec.)
Fibonacci # 23 is : 28657  (Compute time = 1475 Milli Sec.)
Fibonacci # 24 is : 46368  (Compute time = 2386 Milli Sec.)
Fibonacci # 25 is : 75025  (Compute time = 3862 Milli Sec.)
```

The following is the result of running the same program after having been modified to cause a Run Error.

" Output of the Compile/Post_Process step "

```
LINE # P/D LC LVL < Stanford Pascal Compiler, Version of July-78 >

   1    288  1) PROGRAM fib_demo(OUTPUT);
   2    288  1)
   3    288  1) TYPE pos_int = 0..30;
   4    288  1)
   5    288  1) VAR  i     : pos_int;
   6    292  1)      time  : INTEGER;
   7    296  1)
   8    296  1)  FUNCTION fibonacci(j :pos_int) : INTEGER;
   9     84  2)  (*To evaluate fibonacci # j, for j >= 0,
                                       subject to integer overflow*)
  10     84  2)
  11     84  2)   BEGIN
  12      0  2)   IF j = 0 THEN fibonacci := 0
  13      5  2)   ELSE IF j = 1 THEN fibonacci := 1
  14     12  2)     ELSE fibonacci := fibonacci(j-1) + fibonacci(j-3);
  15     29  2)   END;
  16     84  2)
  17     84  2)  BEGIN (*fib_demo*)
  18      0  1)  FOR i := 10 TO 25 DO
  19     15  1)   BEGIN  time := CLOCK(0);
  20     18  1)   WRITELN(' Fibonacci # ', i:3, ' is :', fibonacci(i):6,
  21     37  1)   '  (Compute time =', CLOCK(0)-time:5, ' Milli Sec.)');
  22     53  1)   END;
  23     62  1)  END.


 ****   NO SYNTAX ERROR(S) DETECTED.

 ****   23 LINE(S) READ,  1 PROCEDURE(S) COMPILED,

 ****   94 P_INSTRUCTIONS GENERATED,  0.04 SECONDS IN COMPILATION.
```

```
****   NO ASSEMBLY ERROR(S) DETECTED.

****   672 BYTES OF CODE GENERATED, 0.05 SECONDS IN P_CODE ASSEMBLY.
```

" Output of the GO step "

```
Fibonacci # 10 is :
  **** SNAPSHOT DUMP OF PROGRAM ****

  **** 'SNAPSHOT' was called by --> 'Pascal_MONITOR'
  **** Run Error: 1002 from line: 14 of procedure: 'fibonacci'
  **** SUBRANGE VALUE OUT OF RANGE
  **** The offending value: -1 is not in the range: 0..30

  **** Variables for 'fibonacci' are:

     j   = 2

  **** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

  **** Variables for 'fibonacci' are:

     j   = 3

  **** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

  **** Variables for 'fibonacci' are:

     j   = 4

  **** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

  **** Variables for 'fibonacci' are:

     j   = 5

  **** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

  **** Variables for 'fibonacci' are:

     j   = 6

  **** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

  **** Variables for 'fibonacci' are:

     j   = 7

  **** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14
```

```
**** Variables for 'fibonacci' are:

   j   = 8

**** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

**** Variables for 'fibonacci' are:

   j   = 9

**** procedure 'fibonacci' was called by --> 'fibonacci' from line: 14

**** Variables for 'fibonacci' are:

   j   = 10

**** procedure 'fibonacci' was called by --> '$MAINBLK' from line: 20

**** Variables for '$MAINBLK' are:

   i    = 10
   time  = 25

**** END OF DUMP ****
```

The following is the result of yet another run of the same program with the 'K+' option. The (only) source listing is generated by the last step in the run and it followes any other output that the user program may prooduce. The prototype JCL for this run is provided in section 3(f). (Note the increased "compute" time.)

" Output of the Compile/Post_Process step "

```
****    NO ASSEMBLY ERROR(S) DETECTED.

****    804 BYTES OF CODE GENERATED, 0.06 SECONDS IN P_CODE ASSEMBLY.
```

" Output of the GO step - including the Profiler output "

```
  Fibonacci # 10 is :    55  (Compute time =    3 Milli Sec.)
  Fibonacci # 11 is :    89  (Compute time =    5 Milli Sec.)
  Fibonacci # 12 is :   144  (Compute time =    8 Milli Sec.)
  Fibonacci # 13 is :   233  (Compute time =   13 Milli Sec.)
  Fibonacci # 14 is :   377  (Compute time =   20 Milli Sec.)
  Fibonacci # 15 is :   610  (Compute time =   33 Milli Sec.)
  Fibonacci # 16 is :   987  (Compute time =   54 Milli Sec.)
```

```
Fibonacci # 17 is :  1597  (Compute time =   87 Milli Sec.)
Fibonacci # 18 is :  2584  (Compute time =  140 Milli Sec.)
Fibonacci # 19 is :  4181  (Compute time =  227 Milli Sec.)
Fibonacci # 20 is :  6765  (Compute time =  369 Milli Sec.)
Fibonacci # 21 is : 10946  (Compute time =  595 Milli Sec.)
Fibonacci # 22 is : 17711  (Compute time =  963 Milli Sec.)
Fibonacci # 23 is : 28657  (Compute time = 1562 Milli Sec.)
Fibonacci # 24 is : 46368  (Compute time = 2527 Milli Sec.)
Fibonacci # 25 is : 75025  (Compute time = 4092 Milli Sec.)
```

" Ouput of the PROFILE step "

```
LINE # RUN CNT LVL < Stanford Pascal Compiler, Version of July-78 >


  1           1) (*$K+*)
  2           1) PROGRAM fib_demo(OUTPUT);
  3           1)
  4           1) TYPE pos_int = 0..30;
  5           1)
  6           1) VAR  i     : pos_int;
  7           1)      time : INTEGER;
  8           1)
  9           1)  FUNCTION fibonacci(j :pos_int) : INTEGER;
 10           2)  (*To evaluate fibonacci # j, for j >= 0,
                                     subject to integer overflow*)
 11           2)
 12           2)   BEGIN
 13 121338    2)   IF j = 0 THEN fibonacci := 0
 14 196329    2)   ELSE IF j = 1 THEN fibonacci := 1
 15 317651    2)     ELSE fibonacci := fibonacci(j-1) + fibonacci(j-2);
 16 635318    2)   END;
 17           2)
 18           2)  BEGIN (*fib_demo*)
 19        1  1)  FOR i := 10 TO 25 DO
 20       16  1)   BEGIN  time := CLOCK(0);
 21       16  1)   WRITELN(' Fibonacci # ', i:3, ' is :', fibonacci(i):6,
 22       16  1)    '  (Compute time =', CLOCK(0)-time:5, ' Milli Sec.)');
 23       16  1)   END;
 24        1  1)  END.


 ****   NO SYNTAX ERROR(S) DETECTED.

 ****   24 LINE(S) READ,  1 PROCEDURE(S) COMPILED,

 ****   100 P_INSTRUCTIONS GENERATED,  0.05 SECONDS IN COMPILATION.
```

## 6. Changed Features and New Options

The following list is provided as a convenience to users of previous versions of Stanford Pascal. The list briefly mentions the features that are new or are implemented differently from the earlier versions. These features either correspond to the standard Pascal now, as described in Jensen and Wirth [1], or are described in an earlier section of this document.

-Global textfiles may now be declared and passed as VAR parameters to procedures or functions.

-The character set is now the EBCDIC character set and not the 63 character set that corresponded to the CDC Scientific character set.

-The predefined constant MAXINT, the predefined types ALFA and TEXT, the predefined functions and procedures PAGE, ROUND, LINELIMIT, CARD, SKIP and EXPO are provided.

-The predefined variables DATE, TIME and OSPARM are added.

-The sub-monitor now handles most IBM file formats.

-The sub-monitor now supports input and output of Booleans.

-The sub-monitor now checks the format of Booleans, integers and reals that are input. It also rejects any attempt to read any of these same datatypes when the end of file is reached.

-The sub-monitor will automatically invoke the execution profile generator (PASPROF load module) if the Pascal execution outputs run counts to the QRR file (i.e., if the Pascal program was compiled with the K+ option).

-The input of character strings is now handled differently.

-The JCL parameters passed to the sub-monitor now include NOSNAP, NOSPIE and NOCC.

-User parameter strings may be passed to the Pascal program.

-Comments may be nested (under control of the 'N+' compilation option).

-The M (margins) compilation option has an extended meaning.

-The sequence number field on input cards (col.s 73-80) no longer is printed instead of the source line number when M+ is specified.

-Compilation input is no longer restricted to card image format. The input may contain any of the allowed file formats, but only the first 120 characters in each record are significant.

-The M (margins) compilation option has an extended meaning.

-The sequence number field on input cards (col.s 73-80) no longer is printed instead of the source line number when M+ is specified.

-Compilation input is no longer restricted to card image format. The input may contain any of the allowed file formats, but only the first 120 characters in each record are significant.

-Subranges such as 1..10 are acceptable labels in CASE statements or the variant parts of records. Also subranges may appear in constants of type SET; e.g., [1..4] is equivalent to [1,2,3,4].

-Functions of type SET may be declared.

-The tag field of a case variant record may be left unnamed.

-The offsets of variables in the stack are now assigned differently.

-The first 12 characters of Pascal identifiers are now significant.

-Lower case letters may be used in identifiers and reserved words.

-External Pascal procedures may be created and used.

-FORTRAN subroutines/functions may be called from Pascal programs. (A separate version of the sub-monitor is not required for this.)

-The different versions of the run-time support routine PMONSRC are now merged into a single program and with the use of (boolean) assembly time switches, one may get the compact object form suitable for system programs, or the full sized object to be used in conjunction with user programs.

Acknowledgements

This note owes a great deal to Nigel Horspool of McGill University who, amongst other things, converted a group of chronologically ordered sections into the present document. He also upgraded the I/O interface to provide support for various OS file formats. The SNAPSHOT routine is written by Eral Waldin of SLAC and the run-time profile generator PROFILER is due to Richard Sites of Los Alamos Scientific Laboratory. The programs in the TESTLIB are contributed by many people and it is hoped that it will evolve into a library of utility routines of general interest to Pascal users.

References:

[1] K. JENSEN, N. WIRTH. 'Pascal, User Manual and Report' (2nd ed.), Springer-Verlag, New York, 1975.

[2] K. NORI, U. AMMAN, K. JENSEN, H. NAGEL. 'The Pascal "P" Compiler, Implementation Notes', Berichte des Instituts fur Informatik, E.T.H. Zurich, DEC. 1974.

[3] D. GRIES. 'Compiler Construction for Digital Computers', John Wiley and Sons, New York, 1971.

[4] S. HAZEGHI. 'Bootstrap and Adaptation of a Pascal Compiler on the IBM/370 System', CGTM-194, Stanford Linear Accelerator Center, July 1979.

[5] S. HAZEGHI, L. WANG. 'A Short Note on High Level Languages and Microprocessors', Conference Proceedings of the 2nd West Coast Computer Fair, San Jose, CA., March 1978.

[6] E. GILBERT, D. WALL. 'SOPAIPILLA Maintenance Manual', CSL Technical Report no. 158, Stanford University, March 1978.

[7] B. HITSON. 'Pascal/P_Code Cross Compiler for the LSI-11', SLAC-PUB-2246, Stanford Linear Accelerator Center, Jan. 1979.

Sassan Hazeghi, Nov. 1976.

Computation Research Group,
Stanford Linear Accelerator Center,
Box 4349,
Stanford, CA. 94305.

Phone (415) 854-3300 x2359.

| | |
|---|---|
| Date of last update: | Jan.-26-77. |
| | Mar.-04-77. |
| | May -20-77. |
| | June-09-77. |
| | Nov.-15-77. |
| | Jul.-28-78 |
| | Sep.-18-78 |
| | May -20-79 |
| | July-01-79 |
| | Aug -09-79 |
| | Oct.-18-79 |

Appendix A

1- Pascal compiler error messages:

1- error in simple type.
2- identifier expected.
3- "program" expected.
4- ")" expected.
5- ":" expected.
6- illegal symbol.
7- error in parameter list.
8- "of" expected.
9- "(" expected.
10- error in type.
11- left square bracket expected.
12- right square bracket expected.
13- "end" expected.
14- ";" expected.
15- integer expected.
16- "=" expected.
17- "begin" expected.
18- error in declaration part.
19- error in field list.
20- "," expected.
21- "*" expected.
50- error in constant.
51- ":=" expected.
52- "then" expected.
53- "until" expected.
54- "do" expected.
55- "to" or "downto" expected.
56- "if" expected.
57- "file" expected.
58- error in factor.
59- error in variable.
101- identifier declared twice.
102- low bound exceeds highbound.
103- identifier is not of appropiate class.
104- identifier is not declared.
105- sign not allowed here.
106- number expected.
107- incompatible subrange types.
108- file not allowed here.
109- type must not be real.
110- tagfield type must be scalar or subrange.
111- incompatible with tagfield type.
112- index type must not be real.
113- index type must be scalar or subrange.
114- base type must not be real.
115- base type must be scalar or subrange.
116- error in type of standard procedure parameter.

117- unsatisfied forward reference.
118- forward reference type identifier in variable declaration.
119- forward declared; repetition of parameter list not allowed.
120- function result type must be scalar, subrange, or pointer.
121- file value parameter not allowed.
122- forward declared function; repetion of result type illegal.
123- missing result type in function declaration.
124- f-format is for real type only.
125- error in type of standard function parameter.
126- number of parameters does not agree with declaration.
127- illegal parameter substitution.
128- result type of parm function does not agree with declaratn.
129- type conflict of operands.
130- expression is not of set type.
131- only tests on equality allowed.
132- strict inclusion not allowed.
133- file comparison not allowed.
134- illegal type of operand(s).
135- type of operand must be boolean.
136- set element must be scalar or subrange.
137- set element types not compatible.
138- type of variable is not an array.
139- index type is not compatible with declaration.
140- type of variable is not a record.
141- type of variable must be a file or pointer.
142- illegal parameter substitution.
143- illegal type of loop control variable.
144- illegal type of expression.
145- type conflict.
146- assignment of files not allowed.
147- label type incompatible with selecting expression.
148- subrange bounds must be scalar.
149- index type must not be integer.
150- assignment to standard function is not allowed.
151- assignment to formal function is not allowed.
152- no such field in this record.
153- type error in read.
154- actual parameter must be a variable.
155- control variable may not be declared on intermediate level.
156- multiply defined case label.
157- too many cases in case statement.
158- missing corresponding variant declaration.
159- real or string tagfields not allowed.
160- previous declaration was not forward.
161- duplicate forward declarations.
162- parameter size must be constant.
163- missing variant in declaration.
164- substitution of standard procedure/function not allowed.
165- multidefined label.
166- multideclared label.
167- undeclared label.
168- undefined label.

169- error in base set.
170- value parameter expected.
171- standard file was redeclared.
172- undeclared external file.
173- FORTRAN procedure or function expected.
174- Pascal procedure or function expected.
175- missing file "input" in program heading.
176- missing file "output" in program heading.
177- assignment to function identifier not allowed here.
178- multiply defined record variant.
179- X-opt of actual proc/func does not match formal declaration.
180- control variable must not be formal.
181- constant part of address out of range.
201- error in real constant- digit expected.
202- string constant must not exceed source line.
203- integer constant exceeds range.
204- 8 or 9 in octal number.
205- zero length string not allowed.
206- integer part of real constant exceeds range.
250- too many nested scopes of identifiers.
251- too many nested procedures and/or functions.
252- too many forward references of procedure entries.
253- procedure too long.
254- too many long constants in this procedures.
255- too many errors in this source line.
256- too many external references.
257- too many externals.
258- too many local files.
259- expression too complicated.
260- too many exit labels.
300- division by zero.
301- no case provided for this value.
302- index expression out of bounds.
303- value to be assigned is out of bounds.
304- element expression out of range.
390- premature end of program, (bad program structure).
398- implementation restriction.
399- variable dimension arrays not implemented.
400- illegal expression.
401- compiler consistency check !

## 2- Pascal post-processor error messages:

253- Procedure too long (larger than 8K bytes).
--> Divide (the procedure) and conquer.
254- Too many long (string) constants.
--> Recompile the Post_Processor with a larger value for MXSTR.
256- Too many Procedures/Functions referenced in this Proc.
--> Recompile the Post_Processor with a larger value for MXPRC.
259- Expression too complicated.
--> Simplify the expression by rearranging and/or breaking.
263- Too many (Compiler generated) Labels in this Procedure.
--> Recompile the Post_Processor with a larger value for MXLBL.
281- Too many Integer constants in this Procedure.
--> Recompile the Post_Processor with a larger value for MXINT
282- Too many Double Word (REAL,SET) constants in this Procedure.
--> Recompile the Post_Processor with a larger value for MXDBL.
300- Divide by Zero (result of constant propagation).
--> Fix up the (constant) expression evaluating to Zero.
302- Index/subrange value out of range (constant propagation ?)
--> Fix up the (constant) expression to be within range.
501- Array component too large (larger than 32K).
--> Reduce the range of the last (rightmost) indecies of the array and/or reorder the dimensions of the array so that they are ordered from the largest (leftmost) to the smallest (rightmost).

## Compiler/Post-processor concistancy checks:

601- Type conflict of operands in the P_Program.
602- Operand should be of type 'ADR'.
604- Illegal type for run-time checking.
605- Operand should be of type 'BOOL'.
606- Undefined P_Instruction code.
607- Undefined Standard Procedure name.
608- Displacement field (of address) out of range.
609- Small Proc Larger than 4K.
--> Recompile the Post_Processor with "SHRT_PROC = 300".
611- Bad INTEGER alignment.
612- Bad REAL alignment.
613- Bad REAL constant.
614- Inconsistent Procedure Table file "PRD".
--> Fix the JCL and/or the 'QRR' output of the compiler.

3- Runtime error messages:

| | |
|---|---|
| 1001- | index value out of range. |
| 1002- | subrange value out of range. |
| 1003- | actual parameter out of range. |
| 1004- | set member out of range. |
| 1005- | pointer value invalid. |
| 1006- | stack/heap collision (i.e. program needs mor stak space). |
| 1007- | illegal input/reset operation. |
| 1008- | illegal output/rewrite operation. |
| 1009- | synchronous i/o error. |
| 1010- | program exceeded the specified running time. |
| 1011- | invalid file definition. |
| 1012- | not enough space available. |
| 1013- | undefined or obsolete submonitor call (should not occur). |
| 1014- | LINELIMIT exceeded for output file. |
| 1020- | illegal input past end of file. |
| 1021- | bad BOOLEAN on input. |
| 1022- | bad INTEGER on input. |
| 1023- | bad REAL on input. |
| 200X- | program interruption code 'X', --> enable debug option 'D+' and rerun the program. |
| 3001- | external error (e.g. bad parameter to math routines etc.) |
| X1XX- | unable to call on 'snapshot' after a run error (this happens if there is not enough space or if snapshot was not included in the load module or if the nosnap parameter was specified in jcl) other digits of the return code to be interpreted as above |