# TABLE OF CONTENTS

- **PREFACE** ................................................................. 5

- **SECTION 1: RIDGE PASCAL LANGUAGE NOTES** ................. 7
  - Introduction ......................................................... 7
  - Listing of Differences ........................................... 9
  - Case Statements ................................................... 9
  - Character Synonyms ............................................... 9
  - Comments .................................................................. 10
  - Compiler Options ................................................... 10
  - Declarations .......................................................... 12
  - External Procedures and Functions ................................ 12
  - Files ......................................................................... 13
  - GOTO Statements ..................................................... 15
  - Identifiers ............................................................. 15
  - Mixed Mode Expressions ........................................... 15
  - Numbers .................................................................. 15
  - PACKED Types .......................................................... 17
  - PACK and UNPACK ..................................................... 17
  - Procedures and Functions as Parameters ....................... 17
  - Reserved Words ....................................................... 17
  - String Literals ........................................................ 17
  - Strings ..................................................................... 17
  - Types ....................................................................... 19

- **SECTION 2: THE PASCAL RUNTIME ENVIRONMENT** .......... 21
  - Introduction ........................................................... 21
  - Data Segment Overview ............................................. 23
    - Data Segment Memory Diagrams ................................. 23
      - Absolute Mode .................................................... 23
      - Relocatable Mode ................................................. 25
    - Stack Diagrams ..................................................... 27
    - The Mark Stack Block ............................................. 30
    - The Display ........................................................ 31
    - The Heap ................................................................ 33
  - Code Segment Overview ............................................. 33
    - Code Segment Memory Diagrams ................................. 33
    - Preamble and Postamble Code ................................... 36
      - Procedure/Function Entry Code ............................... 37
      - Procedure/Function Exit Code ................................. 38
      - Program Entry Code .............................................. 40
      - Program Exit Code ................................................ 41
  - Miscellaneous .......................................................... 42
    - Register Use Conventions ........................................ 42
    - Procedure/Function Calling Conventions ..................... 42
    - Data Representation and Alignment Rules .................... 43
This manual documents the Ridge Pascal language, which is based on the standard language as defined by Jensen and Wirth in the "Pascal User Manual and Report." The Ridge language shares various modifications to the base language, including traditional improvements to case statements, character synonyms, comments, and declarations, with other Pascal implementations. These and other changes arose from the desire for performance trade-offs and the need to meet implementation requirements, creating a language suitable for production.

Since a knowledge of Pascal on the part of the reader is assumed, the differences between the Jensen-Wirth language and the Ridge language are documented in this manual but not the Pascal language in its entirety.

This manual is divided into three sections:

- Ridge Pascal Language Notes
- The Pascal Runtime Environment
- An Example

The first section describes Ridge Pascal by listing the differences between it and the Jensen-Wirth language. Topics are arranged alphabetically.

The second section describes the Pascal runtime environment. Much of this information is pictorial; memory diagrams are provided that illustrate the relationships among the various components of a Pascal user process running under the Ridge Operating System (ROS).

The third section gives an example of how to write an assembly language routine that can be called by a Pascal program.
SECTION 1

RIDGE PASCAL LANGUAGE NOTES

INTRODUCTION

This section describes the Ridge Pascal language by citing the differences between it and standard Pascal (as defined in Kathleen Jensen and Nicklaus Wirth's "Pascal User Manual and Report").

The reader is referred to the Jensen/Wirth book, second edition, Springer-Verlag, 1975, for a detailed discussion of the base language.

The following list gives an overview of where Ridge Pascal differs from standard Pascal. The list is in alphabetical order for easy reference, and each item is explained in detail in the remainder of this section.

- Case Statements
- Character Synonyms
- Comments
- Compiler Options
- Declarations
- External Procedures and Functions
- Files
  - EOF and EOLN
  - File Manipulations
    - OpenFile
    - CloseFile
    - FileStatus
  - File Types
o GET
o PUT
o READ
o RESET
o REWRITE
  o Standard Predefined Files
  o WRITE

  o GOTO Statements
  o Identifiers
  o Mixed Mode Expressions
  o Numbers
    o Integers
    o Reals

  o PACKED Types
  o PACK and UNPACK
  o Procedures and Functions as Parameters
  o Reserved Words
  o String Literals
  o Strings
    o How to use
    o NewString

  o Types
LISTING OF DIFFERENCES

Case Statements

In standard Pascal, if there is no case label equal to the value of the case expression, the action of the case statement is undefined. In Ridge Pascal, however, the statement immediately following the case statement is selected for execution.

The case statement has an optional "otherwise" case label. The reserved word "otherwise" may be affixed to the last case alternative rather than a case label, causing control to be transferred to this last alternative in the event of no prior match with other case labels.

Character Synonyms

The following character synonyms are recognized by the Ridge Pascal compiler:

- "|" can be substituted for "or".
- "&" can be substituted for "and".
- "~" can be substituted for "not".

Comments

In Ridge Pascal, the symbols "(\* and \*)" may be used to delimit comments; the standard symbols "{" and "}" may also be used. Comment delimiters must be matched; that is, if a command starts with "{", then it must end with "}"; if it starts with "(\*", then it must end with "\*)". Comments having the the same delimiters may not be nested. All text appearing between delimiters is ignored by the compiler; however, if the first symbol after the first delimiter is "\$", the comment is interpreted as a compiler option (see Compiler Options).
Compiler Options

Compiler options are communicated to the compiler via special comments (see Comments). The following compiler options are recognized by the Ridge Pascal compiler when they follow a "$_" at the beginning of a comment:

- The "E" (eject) option controls pagination of the source listing. The effect is that the next source line will appear at the top of a new page.

- The "G" option controls the starting address of the (static) outer block variables. This option must appear before the "program" declaration. The "G" option implies absolute addressing as opposed to relocatable addressing (see the "R" option).
  - The form of the "G" option is "G<n>" where "<n>" is a decimal integer. For example, "G16384" would cause the compiler to start allocating global variables at 16K.
  - The default is "G4096".

- The "L" option is for source listing control. This option may appear anywhere in the source program.
  - "L+" turns the listing on.
  - "L-" turns the listing off.
  - "L+" is the default.

- The "O" option instructs the compiler whether or not to optimize the object code.
  - "O+" produces optimized object code.
  - "O-" produces unoptimized object code.
  - "O+" is the default.
The "P" option controls the packing of data. It informs the compiler that it should pack data closely, which saves data space but increases execution time. See the Runtime Environment section for information about the layout of data and the effect of packing. This option must appear before the "program" declaration.

- "P+" causes data to be tightly packed.
- "P-" causes nonpacking of data.
- "P-" is the default.

The "R" option causes the compiler to generate code in which the outer block variables are allocated in a relocatable segment rather than being assigned to absolute addresses. This option thus facilitates the construction of a program consisting of a number of separate compilations. With this type of construction, the user will not be burdened with assigning starting addresses for the separate compilations' outer block variables since the linker will perform this task.

Accessing relocatable outer block variables generally causes a slight performance decrease in comparison to accessing absolute outer block variables. The reason for the decrease is that an extra instruction must be executed to determine the base of the separate relocatable compilations' outer block variables.

The "R" option must appear before the "program" declaration. Additionally, it is mutually exclusive with the "G" (outer block variables starting address) and "S" (string constant starting address) options. That is, if the "R" is present, then neither a "G" nor "S" option may appear in the same compilation.
"R+" enables relocatable addressing of global variables.

"R-" disables relocatable addressing of global variables, i.e., causes absolute addressing.

"R-" is the default.

The "S" option controls the starting address from which string constants will be allocated downwards (towards lower addresses). The "S" option implies absolute addressing as opposed to relocatable addressing (see the "R" option). This option must appear before the "program" declaration.

The form of the "S" option is "S<n>" where "<n>" is a decimal integer.

"S0" is the default.

Declarations

LABEL, CONST, and TYPE declarations may appear in any order and may be repeated. However, as in standard Pascal, they may not appear after the first variable, procedure, or function declaration in the current block.

External Procedures and Functions

The "external" attribute is supported for procedures and functions. It is similar to the "forward" attribute in that it tells the compiler that only a procedure heading appears at this point. However, unlike the "forward" attribute which indicates that the body will appear later in the compilation, the "external" attribute indicates that the body has been compiled separately inside another program and will not appear in this compilation. The name of the "external" procedure will be passed on to the linker, which will resolve the reference at link time.

The names of all procedures and functions are considered global and may be referenced by other separately compiled programs.
Files

- EOF(f) and EOLN(f). EOLN is defined as EOF or (f = chr(13)), where "chr(13)" is the ASCII carriage return. Return characters are not, as in standard Pascal, converted to blanks. Nor, unlike standard Pascal, is EOF defined until after the first GET operation.

- File Initialization

   All file variables except the predefined variables "input", "output", and "stderr" must be explicitly opened. There are three file manipulation routines for this purpose, which, since they are not predefined, must be declared as "external." For more information on these routines, see the Ridge "Operating System Reference Manual." The declarations for the routines are as follows (the string type is described later):

   ```pascal
   Procedure OpenFile(
     var f:Text;
     name:String;
     mode:Char
   ); External;
   Function FileStatus(var f:Text):Integer;External;
   Procedure CloseFile(var f:Text);External;
   ```

   - The function of procedure "OpenFile" is to take a Pascal file variable and bind it to the ROS file indicated by the "name" argument. The argument "mode" must be either "R" for read access, "W" for write access, "A" for append access (writing at the end of a file), or "U" for update access (reading or writing).

   - The function "FileStatus" returns the value zero if no errors were encountered during any input/output operation on the file; otherwise, non-zero is returned.

   - The function of procedure "CloseFile" is to release the binding between the Pascal file variable, "f", and the ROS file.
o File Types. Only "Text" files (Text = File of Char) are currently supported.

o GET must only be applied to open files, otherwise the results are undefined. The Ridge Pascal GET differs from standard Pascal in that the file buffer is not defined until the first GET is performed. This facilitates interfacing with interactive files.

o PAGE outputs an ASCII form-feed, i.e., chr(12).

o PUT must only be applied to open files. Ridge Pascal PUT performs as in standard Pascal.

o READ(f, x) is defined as follows:

begin
  GET(f);
  x := f^;
end

while standard Pascal's READ(f, x) is defined as:

begin
  x := f^;
  get(f);
end

o RESET is recognized by the compiler but performs no operation at this time.

o REWRITE is recognized by the compiler but performs no operation at this time.

o Standard Predefined Files.

The files "input", "output", and "stderr" are predefined in the sense that if they appear in the "program" declaration they will be opened automatically and bound to ROS file entities. Specifically, it will appear as if the following statements had been executed, in which "inputName" is a string variable containing the characters "input", "outputName" contains "output", and "stderrName" contains "stderr".

-14-
OpenFile(input, inputName, 'R');
penFile(output, outputName, 'W');
OpenFile(stderr, stderrName, 'W');

- WRITE performs as in standard Pascal.
- WRITELN outputs an ASCII carriage return, i.e., chr(13).

GOTO Statements

GOTO statements may not transfer control out of the current block—jumping out of procedures or functions is not permitted.

Identifiers

Identifiers may be of any length but only the first 16 characters are significant: identifiers which differ only after the sixteenth character position will be regarded as the same identifier. Identifiers must start with an alphabetic character (a letter), but thereafter may contain letters, digits, or underscores. Upper case characters are not distinguished from lower case characters in identifiers.

Mixed Mode Expressions

Ridge Pascal allows mixed mode expressions (e.g., INTEGER and REAL); however, a "var" parameter must be of the same type as the formal parameter.

Numbers

Integer constants in Ridge Pascal differ from standard Pascal in two respects:
- The base (radix) may be specified.
Embedded underscores are allowed for improved readability.

A BNF description of the allowable forms follows:

\[
\begin{align*}
\text{integer number} & \ ::= \text{integer} \mid \text{based integer} \\
\text{integer} & \ ::= \text{digit} \{[\_\_\_] \text{ digit} \}
\text{based integer} & \ ::= \text{base '}' \text{ extended digit} \{[\_\_\_] \text{ extended digit} \}
\text{base} & \ ::= \text{integer} \quad \text{-- base must be in 2..36}
\text{extended digit} & \ ::= \text{letter} \mid \text{digit}
\end{align*}
\]

Here are some examples to illustrate based integers and the use of underscores to improve readability.

\[
\begin{align*}
40\_96 \\
65\_536 \\
2\_147\_483\_647 \quad (* \text{MAXINT} *)
-2\_146\_483\_648 \quad (* \text{MININT} *)
2\#11111111 \\
2\#1111\_1111 \\
8\#377 \\
16\#ff \\
10\#2\_147\_483\_647
\end{align*}
\]

Ridge Pascal supports 32- and 64-bit real numbers, called REAL and DREAL respectively. A double real (DREAL) number is denoted in a fashion similar to the "E" notation except that a "D" or "d" is used instead. For example:

\[
\begin{align*}
pie & = 3.1415926535D0 \\
bignum & = 1.0D250 \\
maxreal & = 6.8056464E38 \\
minreal & = 5.8774728E-39 \\
maxdreal & = 3.595386269724630D308 \\
mindreal & = 1.112536929253601D-308
\end{align*}
\]
PACKED Types

In Ridge Pascal, the reserved word "PACKED" is accepted but has no effect. To control storage allocation, the "p" compiler option is used (see Compiler Options).

PACK and UNPACK

PACK and UNPACK are not currently supported by the Ridge compiler.

Procedures and Functions as Parameters

Ridge Pascal does not allow procedures or functions to be passed as parameters.

Reserved Words

Ridge Pascal treats upper and lower case characters identically in reserved words. The only nonstandard reserved word in Ridge Pascal is "otherwise".

String Literals

Character string literals may be a maximum of 80 characters in length.

Strings

Ridge Pascal does not have a predefined string type. However, the Pascal runtime library supports a string type via routines (these are more fully described in the Ridge "Operating System Reference Manual"). The following example illustrates how strings are currently manipulated.

The example opens a file called "data.x," does some processing, and then closes the file. The procedures and functions in the Pascal runtime library accept and return strings as defined in the type declaration section in the program. The two-step method of allocating an empty string, and then copying the characters
one-by-one into the string, should be employed since string constants cannot be assigned directly to the string.

Also, note that the procedures NewString, OpenFile, FileStatus, and CloseFile are not predefined, and must be declared as external procedures.

Program Example (stderr);

Type
  StringBody = Record
    length : Integer;
    chars : Array[1..1] of Char;
  end;

  String = ^StringBody;

Var
  CharArray : Array[1..6] of Char;
  dataFile : Text;
  fileName : String;
  i : Integer;

Function NewString(length:Integer):String; External;
Procedure OpenFile(var f:Text; name:String; mode:Char); External;
Procedure CloseFile(var f:Text); External;
Function FileStatus(var f:Text); Integer; External;

begin
  charArray := 'data.x';
  fileName := NewString(6);
  for i := 1 to 6 do
    fileName^.chars[i] := charArray[i];
  OpenFile(dataFile, fileName, 'R');

  if FileStatus(dataFile)<>0 then
    WriteIn(stderr,'cannot open data.x');

  { do some processing }

  CloseFile(dataFile);
end.
Types

Ridge Pascal differs from standard Pascal with respect to types in the following ways:

- DREAL (double REAL) is defined in addition to REAL.

- Sets. The maximum number of set elements is limited to 64. In addition, the following restriction applies to set types and set expressions: in "set of l..u" or "[l..u]", "l" and "u" (or ord(l) and ord(u)) must be in the range of zero to 63 inclusive.

The rules governing data allocation and storage alignment for variables of the various types are heavily dependent on the context of the runtime environment, as well as on the "p" compiler option. The section on the Runtime Environment provides complete details on this subject.
SECTION 2
THE PASCAL RUNTIME ENVIRONMENT

INTRODUCTION

This section provides a fairly detailed picture of the environment in which Pascal programs perform their computations. Enough information is given for the user to perform debugging using the bootstrap debugger, RBUG.

The Ridge architecture maintains separate data and code spaces, and this separation forms the basic division of information in this section. The following topics are covered:

- Data Segment Overview
  - Data Segment Memory Diagrams
    - Absolute Mode
    - Relocatable Mode
  - Stack Diagrams
- The Mark Stack Block
- The Display
- The Heap

- Code Segment Overview
  - Code Segment Memory Diagrams
  - Preamble and Postamble Code
    - Procedure/Function Entry Code
    - Procedure/Function Exit Code
Ridge Pascal

- Program Entry Code
- Program Exit Code

- Miscellaneous
  - Register Use Conventions
  - Procedure/Function Calling Conventions
  - Data Representation and Alignment Rules

The following discussion assumes some familiarity with the Ridge architecture. Information on this subject can be found in the Ridge "Processor Reference Manual."
DATA SEGMENT OVERVIEW

Data Segment Memory Diagrams

The following two subsections provide information regarding the modes that affect memory storage: absolute and relocatable.

ABSOLUTE MODE. Figure 1 gives an overview of the data segment of a Pascal user process when the compiler has been instructed to generate absolute addressing code (see Compiler Options). The blocks are not necessarily to scale—there is a very large gap between the top of the stack and the bottom of the heap.

---

(continued on next page)
Figure 1. Data Segment: Absolute Mode
RELOCATABLE MODE. Figure 2 gives an overview of the data segment of a Pascal user process when the compiler has been instructed to generate relocatable addressing code (see Compiler Options).

(continued on next page)
Figure 2. Data Segment: Relocatable Mode
Stack Diagrams

The Pascal runtime stack expands and contracts as procedures are entered and exited. Each time a procedure is invoked, it allocates a new piece of storage, called a stack frame, on top of the stack for its local variables, context information, parameters, and temporaries.

Figures 3 through 6 represent snapshots of the stack at four significant times in a procedure:

- Normal execution of some arbitrary procedure, "p".
- Preparing for a call to another procedure, "q".
- Entering procedure "q".
- Back in procedure "p".

```
\[\text{Figure 3. Normal Execution of a Procedure "p"}\]
```

In Figure 3, some arbitrary procedure "p" is executing. R15, the Frame Pointer, points to the start of the stack frame for procedure "p". All of "p"s references to local data are based on the Frame Pointer.
In Figure 4, procedure "p" is now preparing to call procedure "q(x, y)" by pushing the parameters onto the stack. The Stack Pointer, R14, does not actually move at this time; rather, the parameters are pushed starting at R14+24, thus leaving a gap for "q"s mark stack block.
Figure 5 shows procedure "q" immediately after it has performed its entry code and the following events have taken place:

- R15 ← R14
- R14 ← R14 + <framesize>
- The mark stack block is filled in.

Notice that now "q" will refer to its parameters at "R15+24" and "R15+32," while the caller referred to them at "R14+24" and "R14+32."
Figure 6. Return to Procedure "p"

Figure 6 shows the stack on return from "q". The stack has been returned to the state it was in just prior to the call to "q".

If "q" had been a Pascal function, then register R0 (or the register pair (R0, R1)) would contain the function value.

The Mark Stack Block

The function of the mark stack block is to store information concerning procedure and function invocations. The mark stack block, therefore, makes it possible to restore the runtime environment when a procedure or function returns to its caller.

Figure 7 shows the format of the mark stack block.
The Display

The display is a sixteen word block which starts at location zero. Figure 8 shows the format for the display block.

When the compiler has been directed to generate absolute addressing code, the display resides at absolute virtual location zero. If the compiler is generating relocatable code, then the display resides at location zero relative to the "$DATA" section.
The Heap

In the case of relocatable addressing, the heap starts at the top of the data segment and grows down towards the lower addresses; in the case of absolute addressing, it starts near the top and grows down. The allocation strategy can be described as follows:

- First, if the number of bytes asked for is "$b\$", then round up "$b\$" to the nearest value such that \((b \mod 8) = 0\). This ensures double word alignment for items that follow it.

- Second, if \((b \mod 4096) = 0\) (i.e., requesting a multiple of pages), then align the allocated block on a page boundary. If \((b \mod 4096) \neq 0\), then the requested block will only be aligned on a double word boundary.
CODE SEGMENT OVERVIEW

Code Segment Memory Diagrams

For the purposes of discussion we will assume the following program, "test." A source program compiled by the Pascal compiler is referred to as a "compilation unit."

Program test( ... )

Procedure a( ... )
begin { of a }
... end; { of a }

Procedure b( ... )

Procedure c( ... )

Procedure d( ... )
begin { of d }
... end; { of d }

begin { of c }
... end; { of c }

begin { of b }
... end; { of b }

Procedure e( ... )
begin { of e }
... end; { of e }

begin { of test }
... end: { of test }

Figure 9 shows how the code segment corresponding to "test" would look, and represents the output of one compilation. Execution begins at location zero.
The code segment of a running user process is usually composed of several compilation units which have been consolidated by the "link" program. Figure 10 shows the overall structure of the code segment of a user process.
Figure 10. Overall Code Segment Structure

Note the following points:

- "link file1 file2 file3" was the command used to produce the illustrated process.

- Since the operating system passes control to the user process at location zero, execution will start at the "main" program in "file1."

- The preamble code for "file2" and "file3" is never executed.
Preamble and Postamble Code

The Pascal compiler generates code prior to the "begin" and after the "end" of a program, procedure, or function. This code performs such miscellaneous housekeeping tasks as stack adjustments and parameter manipulations. This section explains this code.

The code which follows is meant to be interpreted as a "macro" notation. The code in the boxes is generated per the Pascal-like compile-time instructions. For example:

```
FOR I := 1 TO 3 DO
    +-------------------------+
    | ADD R0,R0              | -- double R0
    +-------------------------+
```

The above "macro" code would cause the instruction "ADD R0,R0" to be generated three times.

```
IF <condition> THEN
    +-------------------------+
    | <some code>            |
    +-------------------------+
ELSE
    +-------------------------+
    | <some other code>       |
    +-------------------------+
```

TRUE; otherwise, "<some other code>" would be generated.

In general, the "conditions" of the "macros" refer to attributes of the current program, procedure, or function being compiled.

-36-
PROCEDURE/FUNCTION ENTRY CODE. The following code is generated when a "begin" for a procedure or function is encountered.

IF there are calls THEN
  --------------------------------+
  | STORE R11,R14,0                  |
  --------------------------------+
  -- store return address

IF there are no calls or loops THEN
  --------------------------------+
  | MOVE R12,R15                    |
  --------------------------------+
  -- save dynamic link in R12

ELSE
  --------------------------------+
  | STORE R15,R14,8                 |
  --------------------------------+
  -- save dynamic link in stack

IF it's an intermediate level procedure THEN

IF absolute mode addressing THEN
  --------------------------------+
  | LOAD R10,4*(level-1)            |
  | STORE R10,R14,4                 |
  | STORE R14,4*(level-1)           |
  --------------------------------+
  -- load old static link
  -- store it in the stack
  -- store new static link

ELSE
  --------------------------------+
  | LADDR R8,$DATA,L                |
  | LOAD R10,R8,4*(level-1)         |
  | STORE R10,R14,4                 |
  | STORE R14,R8,4*(level-1)        |
  --------------------------------+
  -- load address of $DATA
  -- load old static link
  -- store it in the stack
  -- store new static link

IF absolute addressing OR static level is not 1 THEN

IF absolute addressing OR static level is not 1 THEN
  --------------------------------+
  | MOVE R15,R14                    |
  | LADDR R14,R14,size              |
  --------------------------------+
  -- allocate local stack frame
  -- R15 <-- frame pointer
  -- allocate stack frame
FOR i := 1 TO number_of_parameters DO
    IF non-VAR array or record THEN
        -------------------------------
        | LADDR Rx,Rl5,disp              |
        | LOAD Ry,Rl5,disp                |
        | LADDR R8, -(byte_count)         |
        | LOADB R9,Ry,0                   |
        | STOREB R9,Rx,0                  |
        | ADDI Ry, 1                      |
        | ADDI Rx, 1                      |
        | LOOP R8, 1, *12                 |
        -------------------------------

In the code which manipulates the static link, the "level" refers to the textual level number of this procedure. The main program is considered level one; procedures which are declared at the program level are at level two; procedures inside these are considered level three; etc.

PROCEDURE/FUNCTION EXIT CODE. The following code is generated when an "end" for a procedure or function is encountered by the compiler.

IF it's a function THEN
    -------------------------------
    | LOAD R0,Rl5,16                 |
    -------------------------------

IF it's a two word value THEN
    -------------------------------
    | LOAD R1,Rl5,20                 |
    -------------------------------

IF it's an intermediate level
    procedure THEN
    IF absolute mode addressing THEN
        -------------------------------
        | LOAD R10,Rl5,4                 |
        | STORE R10,4*(level-1)          |
        -------------------------------
    ELSE
        -------------------------------
        | LADDR R8, $DATA, L            |
        | LOAD R10,Rl5,4                |
        | STORE R10, R8, 4*(level-1)    |
        -------------------------------
IF there were calls THEN
| LOAD R11,R15,0          |
+----------------------------+
-- load return address

{ always do this }
+----------------------------+
| MOVE R14,R15              |
+----------------------------+
-- deallocate stack frame

IF there were no (calls or loops) THEN -- restore old R15
+----------------------------+
| MOVE R15,R12              |
+----------------------------+
-- ... from R12

ELSE
+----------------------------+
| LOAD R15,R15,8            |
+----------------------------+
-- ... from stack

{ always do this }
+----------------------------+
| RET R11,R11               |
+----------------------------+
-- return to caller

For an explanation of "level" see the preceding section on Procedure/Function Entry Code.
PROGRAM ENTRY CODE. The first three boxes of code are generated when the compiler encounters the "program" declaration. Then at "$MAINBLK", in response to the "begin" of the main program, the standard Procedure/Function Entry Code is generated, followed by code which is particular to the main program.

IF absolute addressing mode THEN

00000000 | LADDR R10,$HEAP
| STORE R10,64
| MOVEI R14,0

-- load heap start address
-- store into heap pointer
-- initialize R14

ELSE

+-+--------------------------+
| LADDR R14,$STACK,L
| MOVEI R10,0
| STORE R10,64

-- initialize stack pointer
-- R10 <- 0
-- initialize heap pointer

{ always do this }

+-+--------------------------+
| MOVEI R15,0
| BR $MAINBLK

-- initialize frame pointer
-- branch to main program

Code for all local procedures/functions goes here

$MAINBLK:

+-+--------------------------+
| Proc/Func Entry Code

-- do the same as for procedures

IF absolute addressing mode THEN

+-+--------------------------+
| MOVE R15,R14
| LADDR R14,R14,size

-- initialize frame pointer
-- allocate outer block variables

{ always do this }

+-+--------------------------+
| CALL R11,SYSENTRY

-- initialize Pascal RTL

-40-
IF standard "input" file present THEN
  +----------------------------------------+
  | LADDR Rx,132                           |
  | STORE Rx,R14,24                        |
  | CALL R11,PDF                           |
  +----------------------------------------+
  -- load file buffer address
  -- store file buffer address
  -- open the file

IF standard "output" file present THEN
  +----------------------------------------+
  | LADDR Rx,140                           |
  | STORE Rx,R14,24                        |
  | CALL R11,PDF                           |
  +----------------------------------------+
  -- load file buffer address
  -- store file buffer address
  -- open the file

IF standard "stderr" file present THEN
  +----------------------------------------+
  | LADDR Rx,148                           |
  | STORE Rx,R14,24                        |
  | CALL R11,PDF                           |
  +----------------------------------------+
  -- load file buffer address
  -- store file buffer address
  -- open the file

PROGRAM EXIT CODE. The compiler generates the following code when it encounters the "end" of a main program.

{ always do this }
  +----------------------------------------+
  | MOVEI Rx,0                             |
  | STORE Rx,R14,24                        |
  | CALL R11,SYSEXIT                      |
  +----------------------------------------+
  -- 0=successful completion
  -- store R0
  -- program stops, SYSEXIT doesn't return

| proc/func exit code           |
+----------------------------------------+
  -- same as standard exit code
MISCELLANEOUS

This section discusses miscellaneous runtime issues that do not fit readily into one of the preceding categories. These include register use conventions, procedure/function calling conventions, and data representation and alignment rules.

Register Use Conventions

R0 \   
R1   / register stack to
R2   
R3   \ register stack to
R4   / evaluate expressions
R5   
R6   
R7 /
R8 \   
R9   > scratch registers
R10 /
R11 return address register
R12 \   "with" and "for" temporaries
R13 /
R14 Stack Pointer
R15 Frame Pointer

R0 (or the register pair (R0, R1)) is also used to return the result of a function call.

Procedure/Function Calling Conventions

The general rules for a procedure or function call are as follows:

\[ p(p1, p2, \ldots, pN) \]
o Evaluate parameter 1. Store it at R14,24.

o Evaluate parameter 2. Store it at R14,32.

o Evaluate parameter N. Store it at R14,24+(N-1)*8

The process of evaluating a parameter entails the following:

o Code is generated to evaluate the parameter expression.

o Depending on whether or not the corresponding formal parameter is a "var", there are two cases:

  o "var". In this case the parameter's ADDRESS is stored at R14,24+(j-1)*8, where j is the parameter number, 1 <= j <= N.

  o Non-"var". This case is broken down into two subcases depending on whether the actual parameter is an array or a record.

    o The actual parameter is an array or a record. Pass the ADDRESS as described above.

    o The actual parameter is neither an array nor a record. The VALUE of the parameter is passed.

If "p" is a Pascal function (as opposed to a procedure) then the caller will expect to find the function value in either register R0 (or the register pair (R0,R1)).

Data Representation and Alignment Rules

The compiler packing option "P+" or "P-" controls the amount of storage allocated to a variable of the following standard types:

- BOOLEAN: One byte if "P+", four bytes if "P-".

- CHAR: One byte always.
o DREAL: Eight bytes always.

o Enumerated Types: the minimum number of bytes depends on the number of identifiers in the type:
   o One byte for 1 to 255 elements.
   o Two bytes for 256 to 65,535 elements.
   o Four bytes for more than 65,535 elements.

o FILE or TEXT: Eight bytes always. The last byte, i.e., the one with the highest address, is the file variable "f".

o INTEGER: Four bytes always.

o POINTER: Four bytes always.

o REAL: Four bytes always.

o SET: Eight bytes always.

o Subranges:
   o If the packing option is set to "P-" then all subranges occupy four bytes.

   o If the packing option is set to "P+" then the minimum number of bytes is used. This depends on the lower and upper bounds of the subrange, as the following explains:
      o Negative lower bound always results in four bytes.

      o Lower bound of zero or more results in the following:
         o Upper bound of 1 to 255 results in 1 byte.

         o Upper bound of 256 to 65,535 results in 2 bytes.

         o Upper bound that is more than 65,535 results in four bytes.
The rules for Ridge Pascal data alignment are as follows:

- Half-word items must be aligned on a half-word boundary, i.e., their addresses must be evenly divisible by two.

- Word items must be aligned on a word boundary, i.e., their addresses must be evenly divisible by four.

- Double-word items must be aligned on a double-word boundary, i.e., their addresses must be evenly divisible by eight.

To optimize use of space, the preceding rules should be observed. For example, in declaring variables (or fields in a record) the order of the items may have an impact on the total amount of storage used.

```plaintext
ch : Char ; { 1 byte data }
i : Integer ; { 3 bytes padding, 4 bytes data }
b : Boolean ; { 1 byte data }
d : Dreal ; { 7 bytes padding, 8 bytes data }
k : 1..1000 ; { 2 bytes data }
```

Storage would be used more efficiently if the items were arranged as follows:

```plaintext
ch : Char ; { 1 byte data }
b : Boolean ; { 1 byte data }
k : 1..1000 ; { 2 bytes data }
i : Integer ; { 4 bytes data }
d : Dreal ; { 8 bytes data }
```
Declarations of the following sort are also inefficient:

```pascal
    a : Array[1..100000] of Integer ;
    ch : Char ;
    ...
    ***
    i : Integer ;
```

An improvement would be to declare the large array last, then short offsets could be used in the code that accesses "ch", "i", and other scalar variables. Refer to the "Ridge Processor Reference Manual" for more information on this topic.
SECTION 3
AN EXAMPLE

INTRODUCTION

This section illustrates how to write assembly language programs that are Pascal callable. A program written in Ridge Pascal can be compiled into an intermediate form called P-code by the Ridge Pascal compiler, "pasc." The P-code can then be translated into an object module by the translator, "ptrans," and finally linked with other object modules by the linker, "link." (For more information on the compiling process, see the Ridge "Operating System Reference Manual.")

Included in this section are the listings for four files:

- The command file which compiles, assembles, and links the program.
- The Pascal source listing of the main program.
- The assembler listing of the compiler's generated code.
- The assembler listing of the called routines.

The key items to be observed are:

- how the assembler programs are declared in the Pascal program as "external" functions,
- how the assembler programs access their parameters and how they return their values,
- that Pascal compilation is a two step process involving:
  - running the Pascal compiler, "pasc" whose input is "example.s" and whose outputs are "example.l" and "example.p",
  - running the P-code translator, "ptrans" whose input is "example.p" and whose outputs are "example.a" and "example.o".
COMMAND FILE LISTING

pasc -l example.l example.s
ptrans -l example.a example.p
rasm -l asmfuncs.l asmfuncs.s
link -l example.ll example.o asmfuncs.o /lib/rtl.o

PASCAL SOURCE LISTING

{$A+}
{
  This program reads real numbers and computes
  their square roots using Newton's method. Two assembler
  language routines are called to manipulate parts of the
  real numbers.

  The routines are part of a suite of routines defined
  in the book "Software Manual for the Elementary Functions"
}
program example(input, output) ;

var
  z : real ;
  iterations : integer ;

{  'intxp' returns the unbiased exponent of 'x'.
}
function intxp(x : real) : integer ; external ;

{  'setxp' returns a real number whose mantissa is
    that of 'x' and whose exponent is 'n'.
}
function setxp(x : real ; n : integer) : real ; external ;
{  
}$E$
function sqroot(x : real) : real ;

label 99 ;
const
  EPSILON = 1.0E-30;
var
  i : integer;
  yn, ynminusl : real;
begin {* begin of function sqroot ***}
  iterations := 0;
  if x = 0.0 then
    sqroot := 0.0
  else
    begin
      if x < 0.0 then
        x := -x;
      ynminusl := setxp(x, intxp(x) div 2);
      while TRUE do
        begin
          yn := (ynminusl + x/ynminusl) / 2.0;
          iterations := iterations + 1;
          if abs(yn - ynminusl) <= EPSILON then
            goto 99;
          ynminusl := yn;
        end;
      99: sqroot := yn;
    end;
  end; {* end of function sqroot ***}
begin {* begin of program example ***}
  while not eof(input) do
    begin
      readln(input, z);
      writeln(output, 'sqroot(', z, ') = ', sqroot(z),
      ', iterations = ', iterations);
    end;
end. {* end of program example ***}

ASSEMBLER LISTING OF MAIN PROGRAM

SOURCE LINE 41SL=P, ABS_AD=T, $S=0

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Label</th>
<th>Op1</th>
<th>Op2</th>
<th>Func</th>
<th>Mem1</th>
<th>Mem2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>DEA0FFFFFFF</td>
<td>LADDR</td>
<td>R10,-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000006</td>
<td>A6A00040</td>
<td>STORE</td>
<td>R10,64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000A</td>
<td>11E0</td>
<td>MOVEI</td>
<td>R14,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000C</td>
<td>11F0</td>
<td>MOVEI</td>
<td>R15,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
00000000E 9B00FFFFFFFF BR $MAINBLK

SQROOT:
00000014 A7BE0000 STORE R11, R14, 0
00000018 A7FE0008 STORE R15, R14, 8
0000001C 01FE MOVE R15, R14
0000001E DFEFPFFFFFFF LADDR R14, R14, -1
00000024 1100 MOVEI R0, 0
00000026 A6001004 STORE R0, 4100

SOURCE LINE 42
0000002A C71F0018 LOAD R1, R15, 24
0000002E 1120 MOVEI R2, 0
00000030 8A12FFFE BR R1<>R2, E3

SOURCE LINE 44
00000034 1130 MOVEI R3, 0
00000036 A73F0010 STORE R3, R15, 16
0000003A 8B00FFFE BR L4

E3:

SOURCE LINE 46
0000003E 1130 MOVEI R3, 0
00000040 2A13 RCOMP R1, R3
00000042 5510 TESTLT R1, 0
00000044 8E11FFFE BR R1<>1, E5

SOURCE LINE 47
00000048 C74F0018 LOAD R4, R15, 24
0000004C 2254 RNEG R5, R4
0000004E A75F0018 STORE R5, R15, 24

E5:

SOURCE LINE 48
00000052 C70F0018 LOAD R0, R15, 24
00000056 A70E0018 STORE R0, R14, 24
0000005A CFEE0020 LADDR R14, R14, 32
0000005E A70E0018 STORE R0, R14, 24
00000062 93B0FFFEFF CALL R11, INTXP
00000068 0180 MOVE R8, R0
0000006A 5580 TESTLT R8, 0
0000006C 0308 ADD R0, R8
0000006E 7301 ASRI R0, 1
00000070 A70E0000 STORE R0, R14, 0
00000074 CEEFPEFE0 LADDR R14, R14, -32
00000078 93B0FFFEFF CALL R11, SETXP
0000007E A70F0028 STORE R0, R15, 40
00000082 01C0 MOVE R12, R0

VARIABLE AT 2, 40 ASSIGNED TO REGISTER 12
00000084 C71F0018 LOAD R1, R15, 24
00000088 01D1 MOVE R13, R1

VARIABLE AT 2, 24 ASSIGNED TO REGISTER 13

W7:

SOURCE LINE 49
SOURCE LINE 51
0000008A 010C MOVE R0, R12
0000008C 011D MOVE R1, R13

-50-
0000008E 0120 MOVE R2,R0
00000090 2612 RDIV R1,R2
00000092 2310 RADD R1,R0
00000094 DE304000000 LADDR R3,1073741824
0000009A 2613 RDIV R1,R3
0000009C A71F0024 STORE R1,R15,36

SOURCE LINE 52
000000A0 C6401004 LOAD R4,4100
000000A4 1341 ADDI R4,1
000000A6 A6401004 STORE R4,4100

SOURCE LINE 53
000000AA 2410 RSUB R1,R0
000000AC 7011 LSLI R1,1
000000AE 7111 LSRI R1,1
000000B0 DE500DA2460 LADDR R5,228737632
000000B6 2A15 RCOMP R1,R5
000000B8 5C10 TESTLE R1,0
000000BA 8E11FFFF BR R1<1,E9

SOURCE LINE 54
000000BE A7CF0028 STORE R12,R15,40
000000C2 A7DF0018 STORE R13,R15,24
000000C6 8B00FFFF BR X2

E9:
L10:
SOURCE LINE 55
000000CA C70F0024 LOAD R0,R15,36
000000CE 01C0 MOVE R12,R0

SOURCE LINE 56
000000D0 8B00FFBB BR W7

L8:
000000D4 A7CF0028 STORE R12,R15,40
000000D8 A7DF0018 STORE R13,R15,24

X2:
SOURCE LINE 57
000000DC C70F0024 LOAD R0,R15,36
000000E0 A70F0010 STORE R0,R15,16

L4:
SOURCE LINE 59
000000E4 C70F0010 LOAD R0,R15,16
000000E8 C7BF0000 LOAD R11,R15,0
000000EC 01EF MOVE R14,R15
000000EE C7FP0008 LOAD R15,R15,8
000000F2 57BB RET R11,R11

SOURCE LINE 63
$MAINBLK:
000000F4 A7BE0000 STORE R11,R14,0
000000F8 A7FE0008 STORE R15,R14,8
000000FC 01FE MOVE R15,R14
000000FE DFEFFFFFFFF LADDR R14,R14,-1
00000104 93BFFFFFFFFFF CALL R11,SYSENTRY
0000010A CE00008C LADDR R0,140
0000010E A70E0018 STORE R0,R14,24

-51-
Ridge Pascal

Reference Manual

00000112 93B0FF0000F0 CALL R11, PDF
00000118 CE100084 LADDR R1, 132
0000011C A71E0018 STORE R1, R14, 24
00000120 93B0FF0000F2 CALL R11, PDF

W4:
00000126 CE000084 LADDR R0, 132
0000012A C710FFFC LOAD R1, R0, -4
0000012E C7100000 LOAD R1, R1, 0
00000132 7811 CSLI R1, 1
00000134 0181 MOVE R8, R1
00000136 7811 CSLI R1, 1
00000138 0918 OR R1, R8
0000013A 1B11 ANDI R1, 1
0000013C 8611000F BR R1 = 1, L5

SOURCE LINE 65
00000140 CE200084 LADDR R2, 132
00000144 CE301000 LADDR R3, 4096
00000148 A72E0018 STORE R2, R14, 24
0000014C A73E0020 STORE R3, R14, 32
00000150 93B0FF0000F0 CALL R11, RDR
00000156 C70E0018 LOAD R0, R14, 24
0000015A A70E0018 STORE R0, R14, 24
0000015E 93B0FF0000F0 CALL R11, RLN

SOURCE LINE 66
00000164 CE10008C LADDR R1, 140
00000168 CE2000FF LADDR R2, 8
0000016C DB80737126F LADDR R8, 1936814703
00000172 A6800FF8 STORE R8, 8
00000176 DB806F742827 LADDR R8, 1869883431
0000017C A680FFF6 STORE R8, 4
00000180 1137 MOVEI R3, 7
00000182 1147 MOVEI R4, 7
00000184 A71E0018 STORE R1, R14, 24
00000188 A72E0020 STORE R2, R14, 32
0000018C A73E0028 STORE R3, R14, 40
00000190 A74E0030 STORE R4, R14, 48
00000194 93B0FF0000F0 CALL R1, WRS
00000198 C70E0018 LOAD R0, R14, 24
0000019E C6100100 LOAD R1, 4096
000001AA 112E MOVEI R2, 14
000001A2 1130 MOVEI R3, 0
000001A6 A70E0018 STORE R0, R14, 24
000001AA A71E0020 STORE R1, R14, 32
000001AE A72E0028 STORE R2, R14, 40
000001B2 A73E0030 STORE R3, R14, 48
000001B6 93B0FF0000F0 CALL R11, WRR
000001BC C70E0018 LOAD R0, R14, 24
000001C0 CE1000FF LADDR R1, -12
000001C4 DB8029203D20 LADDR R8, 689978656
000001CA A680FF4 STORE R8, -12
000001CE 1124 MOVEI R2, 4
000001D0 1134 MOVEI R3, 4

-52-
<table>
<thead>
<tr>
<th>Decimal Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001D2</td>
<td>A70E0018</td>
<td>STORE R0,R14,24</td>
</tr>
<tr>
<td>000001D6</td>
<td>A71E0020</td>
<td>STORE R1,R14,32</td>
</tr>
<tr>
<td>000001DA</td>
<td>A72E0028</td>
<td>STORE R2,R14,40</td>
</tr>
<tr>
<td>000001DE</td>
<td>A73E0030</td>
<td>STORE R3,R14,48</td>
</tr>
<tr>
<td>000001E2</td>
<td>93B0FFFFFFFB2</td>
<td>CALL R11,WRS</td>
</tr>
<tr>
<td>000001E8</td>
<td>C70E0018</td>
<td>LOAD R0,R14,24</td>
</tr>
<tr>
<td>000001EC</td>
<td>A70E0000</td>
<td>STORE R0,R14,0</td>
</tr>
<tr>
<td>000001F0</td>
<td>13E8</td>
<td>ADDI R14,8</td>
</tr>
<tr>
<td>000001F2</td>
<td>C6101000</td>
<td>LOAD R1,4096</td>
</tr>
<tr>
<td>000001FE</td>
<td>A71E0018</td>
<td>STORE R1,R14,24</td>
</tr>
<tr>
<td>000002FA</td>
<td>83B0FE1B</td>
<td>CALL R11,SQROOT</td>
</tr>
<tr>
<td>000001FE</td>
<td>11E</td>
<td>MOVEI R1,14</td>
</tr>
<tr>
<td>00000200</td>
<td>1120</td>
<td>MOVEI R2,0</td>
</tr>
<tr>
<td>00000202</td>
<td>C73EFFF8</td>
<td>LOAD R3,R14,-8</td>
</tr>
<tr>
<td>00000206</td>
<td>14E8</td>
<td>SUBI R14,8</td>
</tr>
<tr>
<td>00000208</td>
<td>A73E0018</td>
<td>STORE R3,R14,24</td>
</tr>
<tr>
<td>0000020C</td>
<td>A70E0020</td>
<td>STORE R0,R14,32</td>
</tr>
<tr>
<td>00000210</td>
<td>A71E0028</td>
<td>STORE R1,R14,40</td>
</tr>
<tr>
<td>00000214</td>
<td>A72E0030</td>
<td>STORE R2,R14,48</td>
</tr>
<tr>
<td>00000218</td>
<td>93B0FFFFFF9E</td>
<td>CALL R11,WRR</td>
</tr>
<tr>
<td>0000021E</td>
<td>C70E0018</td>
<td>LOAD R0,R14,24</td>
</tr>
</tbody>
</table>

**SOURCE LINE 67**

<table>
<thead>
<tr>
<th>Decimal Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000222</td>
<td>188F</td>
<td>NOTI R8,15</td>
</tr>
<tr>
<td>00000224</td>
<td>CE10FFE4</td>
<td>LADDR R1,-28</td>
</tr>
<tr>
<td>00000228</td>
<td>E79E0024</td>
<td>LOAD R9,R8,36</td>
</tr>
<tr>
<td>0000022C</td>
<td>B79EFFFFFFF4</td>
<td>STORE R9,R8,-12</td>
</tr>
<tr>
<td>00000232</td>
<td>8784FFF7</td>
<td>LOOP R8,4,*-10</td>
</tr>
<tr>
<td>00000236</td>
<td>8B000016</td>
<td>BR</td>
</tr>
<tr>
<td>0000024C</td>
<td>112F</td>
<td>MOVEI R2,15</td>
</tr>
<tr>
<td>0000024E</td>
<td>113F</td>
<td>MOVEI R3,15</td>
</tr>
<tr>
<td>00000250</td>
<td>A70E0018</td>
<td>STORE R0,R14,24</td>
</tr>
<tr>
<td>00000254</td>
<td>A71E0020</td>
<td>STORE R1,R14,32</td>
</tr>
<tr>
<td>00000258</td>
<td>A72E0028</td>
<td>STORE R2,R14,40</td>
</tr>
<tr>
<td>0000025C</td>
<td>A73E0030</td>
<td>STORE R3,R14,48</td>
</tr>
<tr>
<td>00000260</td>
<td>93B0FFFFFFF82</td>
<td>CALL R11,WRS</td>
</tr>
<tr>
<td>00000266</td>
<td>C70E0018</td>
<td>LOAD R0,R14,24</td>
</tr>
<tr>
<td>0000026A</td>
<td>C6101004</td>
<td>LOAD R1,4100</td>
</tr>
<tr>
<td>0000026E</td>
<td>112C</td>
<td>MOVEI R2,12</td>
</tr>
<tr>
<td>00000270</td>
<td>A70E0018</td>
<td>STORE R0,R14,24</td>
</tr>
<tr>
<td>00000274</td>
<td>A71E0020</td>
<td>STORE R1,R14,32</td>
</tr>
<tr>
<td>00000278</td>
<td>A72E0028</td>
<td>STORE R2,R14,40</td>
</tr>
<tr>
<td>0000027C</td>
<td>93B0FFFFFFF</td>
<td>CALL R11,WRI</td>
</tr>
<tr>
<td>00000282</td>
<td>C70E0018</td>
<td>LOAD R0,R14,24</td>
</tr>
<tr>
<td>00000286</td>
<td>A70E0018</td>
<td>STORE R0,R14,24</td>
</tr>
<tr>
<td>0000028A</td>
<td>93B0FFFFFFF</td>
<td>CALL R11,WLN</td>
</tr>
</tbody>
</table>

**SOURCE LINE 68**

<table>
<thead>
<tr>
<th>Decimal Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000290</td>
<td>8B000E97</td>
<td>BR W4</td>
</tr>
</tbody>
</table>

**L5:**

**SOURCE LINE 70**

<table>
<thead>
<tr>
<th>Decimal Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000294</td>
<td>1100</td>
<td>MOVEI R0,0</td>
</tr>
<tr>
<td>00000296</td>
<td>A70E0018</td>
<td>STORE R0,R14,24</td>
</tr>
<tr>
<td>0000029A</td>
<td>93B0FFFFFFF</td>
<td>CALL R11,SYSEXIT</td>
</tr>
</tbody>
</table>
ASSEMBLER LISTING OF CALLED ROUTINES

$HEXOUT

function intxp(x : real) : integer ;

INTXP returns the unbiased exponent of the given
argument, i.e. returns (exponent - 127).

input : R14,24      -- x

output: R0          -- the answer

GLOBAL  INTXP

INTXP:
LOAD  R0,R14,24    ;R0 <<- REAL NUMBER, I.E., LOAD x
CSLI  R0,9         ;SHIFT EXPONENT INTO POSITION
LADDR R1,0FFH      ;LOAD MASK
AND   R0,R1        ;MASK OUT MANTISSA AND SIGN BIT
LADDR R0,127       ;LOAD EXPONENT BIAS
SUB   R0,R1        ;UNBIAS EXPONENT
RET   R11,R11      ;RETURN TO CALLER


function setxp(x : real ; n : integer) : real ;

SETXP returns the real whose mantissa is that of x
and whose exponent is n.

input:  R14,24      -- x
         R14,32      -- n, unbiased exponent

output: R0          -- the answer

GLOBAL  SETXP

SETXP:
LOAD  R0,R14,24    ;R0 <<- REAL NUMBER, I.E., LOAD x
LADDR R1,0807FFFFFH,L ;LOAD MASK
AND   R0,R1        ;CLEAR EXPONENT
LOAD  R1,R14,32    ;LOAD EXPONENT, I.E. LOAD n
LADDR R2,127       ;LOAD EXPONENT BIAS
ADD   R1,R2        ;ADD IN EXPONENT BIAS
LADDR R2,0FFH
AND R1,R2
CSLI R1,15
CSLI R1,8
OR R0,R1
RET R11,R11

;LOAD MASK
;ISOLATE 8-BIT EXPONENT
;SHIFT INTO POSITION
;... IN TWO SHIFTS
;'OR' IN NEW EXPONENT
;RETURN TO CALLER

; END OF SOURCE FILE
;
END