What's in HP Pascal: A Systems Programming Language

Sue Kimura

Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino, California 95014

Introduction

HP Pascal has been enhanced to include features which allow it to be a systems programming language. These features have made it possible to write the MPE/XL operating system in HP Pascal. The HP Pascal compilers are available on Hewlett-Packard Precision Architecture (HPPA) systems. 1

Historically, Pascal has had a reputation as a student's language. It is known for its structured constructs and strict typing rules. There is no doubt that its structured constructs make it attractive as a programming language. Its strict typing rules, however, while helping the programmer avoid run-time problems, have made it difficult for it to be used as a systems programming language.

This paper focuses on the following systems language features:

```
New Data Representation
Type Coercion
Generic Pointers
Procedure and Function Extensions
Dynamic Routines
Exception Handling
Move Routines
Building Intrinsic Files
```

While descriptions and examples of these features are given, this paper is not a tutorial. The *HP Pascal Reference Manual* and *HP Pascal Programmer's Guide* are available for complete explanations of these features.

To permit access to these system language features, either of the compiler options `standard_level 'hp_modcal'` or `standard_level 'ext_modcal'` is required.

1HP Pascal is a superset of the ANSI/IEEE770X3.97-1983 and ISO 7185:1983 standards.

New Data Representation

A short integer data type, generic pointer types, procedure and function types, and a crunched attribute have been added to HP Pascal.

Shortint

A predefined, short integer data type is available in HP Pascal. `Shortint` is a 16-bit, 2-byte aligned data type. Note that shortint is not the same as the subrange -32768..32767 which in HP Pascal is a 32-bit, 4 byte-aligned data type.

Its purpose is to handle compatibility with the MPE/V operating system. It is analogous to the SPL/V integer data type.

The shortint data type does not require the compiler option `standard_level 'hp_modcal'` or `standard_level 'ext_modcal'`.

Localanyptr, Globalanyptr, Anyptr

Another set of new data types are the generic pointer types: `localanyptr`, `globalanyptr`, and `anyptr`. We will discuss these pointer types later, under the topic **Generic Pointers**.

Procedure and Function Types

Procedure and function types are used to define routines which are dynamically invoked at run-time. We will discuss these types later, under the topic **Dynamic Routines**.

Crunched Structures

In addition to allowing packed structures, HP Pascal allows bit packing of data with crunched structures. In this form of data representation no bits are wasted. This allows the programmer to have the greatest control in determining the layout of data.

The crunched attribute in a structure declaration overrides the alignment restriction for allowed types. The allowed types are integer, shortint, boolean, char, enumeration, and subrange of integer, boolean, char, and enumeration. Crunched structures (e.g., array, record, set) of these types are also allowed.

For example, an integer is 4-byte aligned in an unpacked or packed record. In a crunched record it is bit-aligned.

Note the difference in the data representation of the following records, which are unpacked, packed and crunched:

```
unpacked_record = RECORD
    f1 : 0..7;                {1 byte,  1-byte aligned}
    f2 : 0..255;              {1 byte,  1-byte aligned}
    f3 : 0..65535;            {2 bytes, 2-byte aligned}
    f4 : -32768..32767;       {4 bytes, 4-byte aligned}
    f5 : shortint;            {2 bytes, 2-byte aligned}
    f6 : integer;             {4 bytes, 4-byte aligned}
    END;
       {total size = 16 bytes, record alignment = 4-bytes}

 packed_record = PACKED RECORD

    f1 : 0..7;                {3 bits,  1-bit aligned}
    f2 : 0..255;              {1 byte,  1-bit aligned}
    f3 : 0..65535;            {2 bytes, 1-bit aligned}
    f4 : -32768..32767;       {2 bytes, 1-bit aligned}
    f5 : shortint;            {2 bytes, 2-byte aligned}
    f6 : integer;             {4 bytes, 4-byte aligned}
    END;
       {total size  = 12 bytes, record alignment = 4-bytes}

 crunched_record = CRUNCHED RECORD
    f1 : 0..7;                {3 bits,  1-bit aligned}
    f2 : 0..255;              {1 byte,  1-bit aligned}
    f3 : 0..65535;            {2 bytes, 1-bit aligned}
    f4 : -32768..32767;       {2 bytes, 1-bit aligned}
    f5 : shortint;            {2 bytes, 1-bit aligned}
    f6 : integer;             {4 bytes, 1-bit aligned}
    END;
       {total size = 91 bits, record alignment = 1-bit}
```

A crunched record is most useful when the programmer needs to control the layout of data. For example, he may need to copy the data layout of other machines. However, accessing data when they are not aligned on byte-boundaries is costly. Obviously, it is a space over performance tradeoff.

Type Coercion

Type coercion is a mechanism for circumventing the strict typing rules of Pascal. It is enabled by the compiler option `type_coercion`.

Type coercion allows one type of data to be represented as another type. The type of the expression being coerced is called the source type, and the type the expression is being coerced to is called the target type.

The syntax of type coercion is identical to that of a function call:

```
target_type (source_expression)
```

Note the term source_expression. This term indicates that type coercion may not be used on the left-hand side of an assignment statement.

There are five levels of type coercion. In order of decreasing restrictiveness these levels are:

```
conversion
structural
representation
storage
noncompatible
```

Conversion type coercion is of two types: ordinal type conversion and pointer type conversion.

Ordinal type conversion is used to convert an ordinal type (integer, shortint, enumeration, boolean, char, subrange) to another ordinal type. It is most useful when converting from an enumerated type to an integer type and vice versa. Range checking is done to insure that the value of the source expression is within the range of the target type.

Example

```
$standard_level 'ext_modcal'$
$type_coercion 'conversion'$
PROGRAM ordinal_type_coercion;
TYPE
   spectrum = (red, orange, yellow, green, blue, violet);
VAR
   rainbow : spectrum;
   i       : integer;
BEGIN
rainbow := orange;
i := integer (rainbow);   {i := 1}
i := i + 1;
rainbow := spectrum (i);  {rainbow := yellow}
END.
```

Pointer type conversion is used to change from one pointer type to another pointer type. It may be a short-to-short, short-to-long, long-to-long, or long-to-short pointer conversion. Long-to-short pointer conversion may cause a run-time range error. We will discuss short and long pointers later, under the topic **Generic Pointers**.

The remaining levels of coercion may be viewed as the overlaying of storage of tagless variants within a record. This form of coercion is also called free union coercion. Unlike conversion type coercion, no range checking is done.

The differences in these levels are based on the restrictions regarding the storage allocated for the source and target types, their alignment and their type compatibility (Table 1). The specific rules for type compatibility are described in the *HP Reference Manual*.

Level of Coercion	Storage	Alignment	Type Compatibility
Structural	S = T	S = T	Compatible
Representation	S = T	NR	NR
Storage	S >= T	NR	NR
Noncompatible	NR	NR	NR

S = Source Type
T = Target Type
NR = No Restriction

Table 1. *Restrictions for Free Union Coercion*

Structural type coercion, the most restrictive form of free union coercion, requires that the storage and alignment of the source and target type be the same. Their types must also be compatible. If the source and target types are structures, their component types must also follow these rules.

Example

```
$standard_level 'ext_modcal'$
$type_coercion 'structural'$
PROGRAM structural_type_coercion;
TYPE
   arrtype1 = ARRAY [1..10] OF integer;
   arrtype2 = ARRAY [1..10] OF minint..maxint;
CONST
   ca2 = arrtype2 [10 OF -1000];
VAR
   a1 : arrtype1;
   a2 : arrtype2;
BEGIN
a2 := ca2;             {a2[1] := -1000, ..., a2[10] := -1000}
a1 := arrtype1 (a2);   {a1[1] := -1000, ..., a1[10] := -1000}
END.
```

Structural type coercion is used to assign a2 to a1. It is allowed because both a1 and a2 are arrays with elements that have the same size (32 bits), have the same alignment (4-byte), and are type compatible. The result of the assignment is that each element of a1 has the value -1000.

The remaining three levels do not have alignment and type restrictions. Representation type coercion requires the same storage for the source and target types. Storage type coercion requires the target type to be the same size or smaller than the source type.

Both representation and storage type coercion guarantee that no undefined bits in the target type are accessed. However, undefined bits in the source type may still be accessed. This may occur if the source type has undefined bits because of its packing.

Example

```
$standard_level 'ext_modcal'$
PROGRAM representation_and_storage_type_coercion;
TYPE
   rectype1 = RECORD
      f1 : integer;     {4 bytes, 4-byte aligned}
      END;
         {total size = 4 bytes, alignment = 4 bytes}

   rectype2 = RECORD
      f1 : shortint;    {2 bytes, 2-byte aligned}
      f2 : shortint;    {2 bytes, 2-byte aligned}
      END;
         {total size = 4 bytes, alignment = 2-bytes}

   rectype3 = RECORD
      f1 : boolean;     {1 byte,  1-byte aligned}
      f2 : boolean;     {1 byte,  1-byte aligned}
      f3 : shortint;    {2 byte,  2-byte aligned}
      f4 : shortint;    {2 byte,  2-byte aligned}
      END;
         {total size = 6 bytes, alignment = 2-bytes}
CONST
   cr2 = rectype2 [f1: 0, f2: 1];
   cr3 = rectype3 [f1: false, f2: false, f3: 3, f4: -32768];
VAR
   r1 : rectype1;
   r2 : rectype2;
   r3 : rectype3;
BEGIN
r2 := cr2;                              {initialize r2}
r3 := cr3;                              {initialize r3}
$type_coercion 'representation'$
r1 := rectype1 (r2);                    {r1.f1 := 1}
$type_coercion 'storage'$
r1 := rectype1 (r3);                    {r1.f1 := 3}
END.
```

Representation type coercion is used to assign r2 to r1. Both r1 and r2 are records and take the same amount of storage. Note, however, that r1 and r2 do not have the same alignment; r1 is 4-byte aligned while r2 is 2-byte aligned. The result of the assignment is that r1.f1 has the value 1.

Storage type coercion is used to assign r3 to r1. R3 is larger than r1; consequently, any bits not defined for the type rectype1 is not accessible to r1. Specifically, r3.f4 is not accessible to r1. The result of the assignment is that r1.f1 has the value 3.

Finally, noncompatible type coercion allows any type to be coerced to any other type. As the least restrictive form, it is the most dangerous to use.

Example

```
$standard_level 'ext_modcal'$
PROGRAM noncompatible_type_coercion;
TYPE
   rectype1 = RECORD
      f1 : integer;     {4-bytes, 4-byte aligned}
      END;
   rectype4 = RECORD
      f1 : boolean;     {1-byte, 1-byte aligned}
      END;
VAR
   r1 : rectype1;
   r4 : rectype4;
BEGIN
r4.f1 := false;
$type_coercion 'noncompatible'$
r1 := rectype1 (r4);              {r1.f1 := ??}
END.
```

Noncompatible type coercion is used to assign r4 to r1. Because r4 is smaller than r1, r1.f1 accesses bits not defined for r4. The result of the assignment is a garbage value in r1.f1.

As shown in the above examples, the general rule when using type coercion is obvious: use the most restrictive form of coercion that gets the job done.

Note also that type coercion is applicable at the statement level. Only statements that need type coercion should be bracketed with the appropriate level. A common method used to bracket a type coercion statement is to use the compiler options push and pop:

```
$push, type_coercion 'representation'$
r1 := rectype1 (r3);
$pop$
```

Generic Pointers

Generic pointers are different from the typed pointers in Pascal which manipulate the heap. They are true addresses.

There are two types of generic pointers on a HPPA system. A `long pointer` can point to any addressable object on a HPPA system. A `short pointer` points to a subset of these addressable objects.

HP Pascal defines three pointer types: `localanyptr`, `globalanyptr` and `anyptr`. `Localanypointer` is a 32-bit or short pointer. `Globalanyptr` is a 64-bit or long pointer. `Anyptr` on a HPPA system is a globalanyptr. Since the definition of anyptr may change from system to system, it is wise to use `localanyptr` if a short pointer is desired, or `globalanyptr` if a long pointer is desired.

A long pointer is created using the compiler option `extnaddr` in a type, variable, or formal parameter declaration. Long pointers are primarily used by the operating system and subsystems. Users do not normally need to use long pointers.

Generic pointers are assignment compatible with any other pointer type. Their primary restriction is that they may not be dereferenced. In other words, to access data, a generic pointer must be assigned or coerced to a typed pointer.

Two predefined routines allow the manipulation of these pointers. The predefined function `addr` creates a reference to data. The address returned may point to data in the heap, or to local or global data.

The predefined function `addtopointer` allows for arithmetic manipulation of an address. `Addtopointer` returns a pointer value that is a programmer-specified number of bytes away from the current pointer value.

The preferred way to perform address manipulation is to use these generic pointers and predefined routines, rather than to use tagless variant records. Using these routines allows the HP Pascal compiler to generate more optimal code.

The following is an example of walking through an array and printing out contents of its elements:

Example

```
$standard_level 'ext_modcal'$
PROGRAM generic_pointer (output);
TYPE
   intarrtype = ARRAY [1..20] of integer;
   iptrtype =  ^integer;
CONST
   cintarr = intarrtype [20 of 0];
VAR
   intarr : intarrtype;
   ptr,
   endptr : localanyptr;
   iptr   : iptrtype;
BEGIN
   {initialize elements of intarr to 0}
intarr := cintarr;
   {determine the starting address of intarr}
ptr := addr (intarr);
   {determine the ending address of intarr}
endptr := addtopointer (ptr, sizeof (intarr));
WHILE ptr <> endptr DO
   BEGIN  {print next element}
   iptr := ptr;
   writeln (iptr^);
   ptr := addtopointer (ptr, sizeof (integer));
   END;  {print next element}
END.
```

In this example, addr is used to set the base address of the array intarr. Addtopointer is used to determine the ending address as well as to determine the address of the next element of intarr. Sizeof is used to obtain the size in bytes of the array intarr and of the type integer. Because ptr is a localanyptr it cannot be dereferenced to access the data in intarr. Consequently, ptr is assigned to a typed pointer, iptr, and iptr is dereferenced.

Procedure and Function Extensions

New features have been added to the mechanism for declaring a routine and its parameters. These include an anyvar reference parameter, and options for providing default values for parameters, for making parameters extensible, and for duplicating routine code.

ANYVAR

A formal parameter may be declared as ANYVAR. An anyvar parameter is a reference parameter that accepts an actual parameter of any type. The data that are passed are treated as the type of the formal parameter. In other words, ANYVAR is a form of noncompatible type coercion.

When a parameter is declared as ANYVAR a byte count representing the size of the actual parameter is also passed along with the address of the actual parameter. This information may be used to insure that storage allocated for the actual parameter is not overwritten, as well as to refrain from accessing undefined storage in the actual parameter. The byte count may only be accessed by calling the predefined function `sizeof`.

The following is an example of copying data from one array to another using an ANYVAR parameter and a VAR parameter.

Example

```
$standard_level 'hp_modcal'$
PROGRAM anyvar_parm;
TYPE
   spac = PACKED ARRAY [1..10] OF char;
   lpac = PACKED ARRAY [1..20] OF char;

VAR
   i : integer;
   sp : spac;
   lp : lpac;

PROCEDURE copy_data (
   ANYVAR fromparm : spac;
   VAR    toparm : spac);
VAR
   i : integer;
BEGIN
i := 1;
WHILE (i <= sizeof (fromparm))
      AND
      (i <= sizeof (toparm)) DO
   BEGIN
   toparm[i] := fromparm [i];
   i := i + 1;
   END;
END;

BEGIN
...
copy_data (i, sp);        {first call}
copy_data (lp, sp);       {second call}
END.
```

In this example, the difference between `fromparm` and `toparm` is that a variable of any type may be passed to `fromparm`, but only a variable of type `spac` may be passed to toparm. Sizeof(fromparm) is called to insure that only the data defined for fromparm is assigned to toparm, as in the first call to copy_data. Sizeof(toparm) is called to insure that toparm does not go beyond its bounds in the case that fromparm is larger than toparm, as in the second call to copy_data.

Uncheckable_anyvar

If the byte count of an anyvar parameter is not needed or desired, the anyvar parameter should be declared as OPTION uncheckable_anyvar. In this case the sizeof function returns the size of the formal parameter, rather than the size of the actual parameter.

Example

```
PROCEDURE copy_data (
   ANYVAR fromparm : spac;
          fromparmlen : integer;
   VAR    toparm : spac )
   OPTION uncheckable_anyvar;
   external c;
```

In this example, OPTION uncheckable_anyvar is used to eliminate the byte count and the caller is responsible for passing the size of fromparm.

OPTION uncheckable_anyvar should be used when declaring non-Pascal routines which do not support ANYVAR, or when declaring Pascal routines which are to be called from non-Pascal routines.

Default Parameters

Initialization of parameters is provided by declaring default parameters. Default parameters allow empty actual parameters to be passed.

Default parameters are declared with OPTION default_parms following a routine parameter list. A value is required for each of the defaulted parameters. A reference parameter is only allowed the default value nil.

In a routine with default parameters, the predefined function haveoptvarparm may be used for a formal reference parameter to determine whether an actual parameter was defaulted or supplied by the caller. For a formal value parameter, there is no way to determine whether an actual parameter was defaulted or supplied.

The following is an example of opening a Pascal textfile using default parameters for the name of the file to be opened and length of the file name.

Example

```
$standard_level 'ext_modcal'$
PROGRAM default_parms;
CONST
   maxlen = 1024;
TYPE
   lenrange = 0..maxlen;
   pac = PACKED ARRAY [1..maxlen] OF char;
VAR
   pacv : pac;
   f    : text;
PROCEDURE open_file (
   VAR  f        : text;
   VAR  filename : pac;
        length   : lenrange
   ) OPTION default_parms (filename := nil,
                           length := 0
                          );
VAR
   i : integer;
   fname : PACKED ARRAY [1..maxlen+1] OF char;
BEGIN
IF (haveoptvarparm (filename))
   AND
   (length > 0) THEN
   BEGIN  {file name has been passed}
   FOR i := 1 TO length DO
     fname[i] := filename[i];
   fname[length+1] := ' ';
   rewrite (f, fname);
   END  {file name has been passed}
ELSE
   rewrite (f, '$stdlist');
END;

BEGIN
pacv := 'xxxxx';
open_file (f,pacv, 5);    {first call}
open_file (f);            {second call}
END.
```

In the above example, the first parameter, f, must be passed because it does not have a default value. The remaining two parameters, filename and length, have default values and do not need be passed by the caller.

The first call to open_file opens the file called 'xxxxx'. The second call opens the standard output file because no parameters were passed for filename and length. The predefined function haveoptvarparm is called to determine if an actual parameter was passed, as in the first call, or defaulted, as in the second call. `Length` is checked to verify that it is at least 1.

Extensible Parameters

A routine may also have extensible parameters. Extensible parameters are those which are not required at the end of a parameter list when the routine is called.

Extensible parameters are declared with OPTION `extensible` *n* following a routine parameter list. The value *n* indicates that the first *n* parameters are required. In other words, these are the non-extensible parameters. The value of *n* may be between 0 and the number of parameters declared for the routine.

In the extensible routine, the predefined function `haveextension` may be used to determine if an extensible parameter has been passed.

The following is also an example of opening a Pascal textfile. In this instance, however, the `extensible`, rather than the `default_parms` option is used.

Example

```
PROGRAM extensible_parameters;
...
PROCEDURE open_file (
   VAR  f        : text;
   VAR  filename : pac;
        length   : len )
   OPTION extensible 1;
VAR
   i : integer;
   fname : PACKED ARRAY [1..maxlen+1] OF char;
BEGIN
IF (haveextension (filename))
   AND
   (haveextension (length)) THEN
   BEGIN
   FOR i := 1 TO length DO
     fname[i] := filename[i];
   fname[length+1] := ' ';
   rewrite (f, fname);
   END
ELSE
   rewrite (f, '$stdlist');
END;

BEGIN
pacv := 'xxxxx';
open_file (f,pacv, 5);    {first call}
open_file (f);            {second call}
END.
```

In this example, the first parameter is non-extensible and must be passed by the caller. The remaining two are extensible and do not need to be passed. In the procedure `open_file` the predefine `haveextension` is called to determine if the extensible parameters have been passed.

Note that the calls to `open_file` are identical to those in the default parameters example.

The `extensible` and `default_parms` options may be used together. The semantics of combining these options are described in the *HP Pascal Programmer's Guide*.

Inlining Routines

Sometimes it is useful to have routines that have very simple bodies, such as routines to push and pop items from a stack. The programmer has a choice of calling a procedure or duplicating the same code in each place it is needed. A procedure call may be more readable and maintainable, but does require more execution overhead than duplicated code.

An inline routine allows code for a routine to be duplicated in the place that it is called. It also allows parameters to be passed to such a routine.

In HP Pascal, an inline routine is declared with OPTION `inline` following the routine parameter list.

The use of inlined routines is a performance-for-space tradeoff. Consequently, these routines should be short and include only the code for the most frequently taken path. Large blocks of code that handle special cases should be made into routines that are called from the inlined routine.

The following are examples of inlined procedures for pushing and popping items from a stack.

<u>Example</u>

```
PROCEDURE push (item : itemtype)
   OPTION inline;
BEGIN  {push}
IF tos = topofstack THEN
   setuperror (stackoverflow)
ELSE
   BEGIN
   tos := tos + 1;
   stack[tos] := item;
   END;
END;  {push}

PROCEDURE pop
   OPTION inline;
BEGIN  {pop}
IF tos = bottomofstack
THEN
   setuperror (stackunderflow)
ELSE
   tos := tos - 1;
END;  {pop}
```

These examples show that the error conditions, stack overflow and stack underflow, are handled by calls to the procedure `setuperror`. The bodies of these procedures are very simple.

Other potential uses of inlined routines include performing operations such as exponentiation and exclusive-or which are not defined in Pascal.

Dynamic Routines

Procedure and Function Types

HP Pascal has been extended to to include procedure and function types. Procedure and function types are used to declare routines which are dynamically invoked at run-time. Procedure and function types are also called routine types.

Routine types are defined in the TYPE section. A routine type has no routine name associated with it. It only has its parameters, if any, in its parameter list.

A routine variable is a variable of a routine type. It is assigned a value by calling the predefined procedure `addr` on an actual routine. The actual routine must have parameters which are congruent to the parameters of the routine type. The rules for congruency are the same as those for procedural and functional parameters and are described in the *HP Pascal Reference Manual.*

In addition, for a function type, the type of the actual function return must be identical to that of the function type.

The predefined procedure `call` is used to invoke an actual procedure. The first parameter to `call` is a procedure variable or the result of the predefine `addr` on the actual procedure. The remaining parameters are the actual parameters corresponding to the parameters declared for the procedure type, if any.

Similarly, the predefined function `fcall` is used to invoke an actual function. The parameters to `fcall` are analogous to those of `call`.

The following is an example using these routine types.

<u>Example</u>

```
PROGRAM procedure_and_function_type;
TYPE
   ptype = PROCEDURE (i : integer);
   ftype = FUNCTION : integer;
VAR
   pvar : ptype;
   i    : integer;
PROCEDURE proc (i : integer); external;
FUNCTION func : integer; external;
BEGIN
pvar := addr (proc);
call (pvar,1);
i := fcall (addr (func));
END.
```

In this example, the type declaration `ptype` declares a procedure type with one value parameter of type `integer`. The type declaration `ftype` declares a function that returns an integer type. It has no parameters. The variable pvar is of type ptype.

Addr is called to create a reference to procedure proc and the value is assigned to the variable pvar. The procedure proc is invoked by the predefine call. The parameters to call are the procedure variable pvar and the value 1 for the integer value parameter of the procedure type ptype.

The function func is invoked by the predefine fcall. The parameter to fcall is the result of addr applied to the function func. There are no other parameters because the function type ftype has no parameters.

Unresolved Routines

HP Pascal allows a routine to remain unresolved through the link and load process.[2] At runtime, the predefine addr may be called to determine if an unresolved routine has been resolved. If the routine has been resolved, it may be invoked with the predefines call or fcall.

An unresolved routine is declared with OPTION unresolved following a routine parameter list. The EXTERNAL directive must also be used. It must be a level one routine.

The following is an example of invoking unresolved routines.

2Unresolved routines are not supported on HP-UX systems.

Example

```
$standard_level 'ext_modcal'$
PROGRAM option_unresolved;
VAR
  pvar1 : procedure;
  pvar2 : procedure;
PROCEDURE proc1
   OPTION unresolved;
   external;
PROCEDURE proc2
   OPTION unresolved;
   external;
BEGIN
pvar1 := addr (proc1);
pvar2 := addr (proc2);
IF pvar1 <> nil THEN
   call (pvar1)
ELSE IF pvar2 <> nil THEN
   call (pvar2);
END.
```

In this example there are two unresolved procedures, proc1 and proc2. Neither procedure has any parameters. The predefine addr is called to determine if these procedures are resolved. The check for nil is to verify that addr as returned a valid value.

Exception Handling

When a program is running four forms of exceptions may occur. These forms are: hardware errors, operating system errors, HP Pascal run-time errors, and programmer-defined errors.

In HP Pascal, a TRY-RECOVER block statement has been added to handle these exceptions. On an MPE/V system, the only way to trap runtime exceptions is to use intrinsics such as xlibtrap, xaritrap, and xsystrap.

The TRY-RECOVER construct consists of two parts: the TRY block and the RECOVER statement. In other words, for each TRY block there must be an associated RECOVER statement. A BEGIN END is not needed in the TRY block but is necessary for multiple statements in the RECOVER part.

When executing the statements in the TRY block, execution transfers to the RECOVER statement if an exception is raised. If no exception occurs in the TRY block, execution transfers to the statement following the RECOVER statement.

A user-defined exception is raised by calling the predefined procedure escape with a value for the exception. In the RECOVER statement, the value of the exception is accessed by calling the predefined function escapecode.

The following is an example of a programmer-defined exception:

Example

```
PROCEDURE try_recover (parm : integer);
CONST
   lessthan = 0;
   greaterthan = 1;
TYPE
   small = 0..10;
VAR
   local : small;
BEGIN
TRY
   IF parm < 0 THEN
      escape (lessthan)
   ELSE IF parm > 10 THEN
      escape (greaterthan)
   ELSE
      local := parm
RECOVER
   CASE escapecode OF
      lessthan : writeln ('< 0');
      greaterthan : writeln ('> 10');
      END;
writeln ('done');
END;
```

In this example, there are two exceptions. One exception is that `parm` is less than 0. The other exception is that `parm` is greater than 10. If either exception is encountered, the call to `escape` causes execution to transfer to the RECOVER statement. If the value of the escape code is `lessthan` the string '< 0' is written, if `greaterthan`, '> 10' is written.

If neither of the exceptions is encountered, the variable `local` is assigned the value of `parm` and execution transfers to the statement after the RECOVER part, namely the `writeln ('done')` statement.

This example can also take advantage of the Pascal run-time range checking if the programmer does not care whether the error was less than 0 or greater than 10.

Example

```
PROCEDURE try_recover (parm : integer);
TYPE
   small = 0..10;
VAR
   local : small;
   escapeval : integer;
BEGIN
TRY
   local := parm;
RECOVER
   BEGIN
   escapeval := escapecode;
   writeln (escapeval);
   END;
END;
```

In this example, the try block contains only the assignment statement. If parm is not within the range 0..10 an HP Pascal run-time exception is raised and the escape code is set. When execution transfers to the RECOVER part, the predefine escapecode accesses the HP Pascal escape code and the its value is written.

Note that in the RECOVER part, the value returned from the predefine escapecode is assigned to a local variable escapeval. This is a necessary precaution because system-level escapes may change the escape code. In this example, the call to writeln results in a system fwrite call which may modify the escape code.

On an MPE/XL system, the run-time escape codes for HP Pascal are available in the file PASESC.PUB.SYS.

The above examples are examples of local escapes. A local escape is an escape invoked within the static scope of the TRY block. In other words, it is an escape invoked within the statements in the TRY block.

Raising exceptions is not limited to local escapes. An escape may occur anywhere within the dynamic scope of the TRY block. That is, an escape may also occur within a routine called from a statement in a TRY block. This form of escape is called a nonlocal escape. Raising an exception in the dynamic scope of the TRY block also causes execution to transfer to the RECOVER statement. When more than one TRY block is active, execution transfers to the innermost RECOVER statement.

Example

```
PROCEDURE try_recover (parm : integer);
CONST
   lt0  = 0;
   gt10 = 1;
   gt5  = 2;
TYPE
   small = 0..10;
VAR
   local : small;
PROCEDURE inner_proc (parm : integer);
   BEGIN
   IF parm > 5 THEN
      escape (gt5);
   END;
BEGIN
TRY
   IF parm < 0 THEN
      escape (lt0)
   ELSE IF parm > 10 THEN
      escape (gt10)
   ELSE
      BEGIN
      inner_proc (parm);
      local := parm
      END
RECOVER
   CASE escapecode OF
      lt0  : writeln ('< 0');
      gt10 : writeln ('> 10');
      gt5  : writeln ('> 5');
      END;
writeln ('done');
END;
```

In the procedure inner_proc an exception is raised if parm is greater than 5. In this case the assignment of parm to local in the TRY block does not occur, and execution transfers to the recover statement which handles three exceptions. If parm is in the range 1..4, no exception occurs and execution continues at the assignment statement of parm to local and then jumps to the writeln (done) statement following the RECOVER statement.

Move Routines

There are three predefined procedures, move_fast, move_l_to_r, and move_r_to_l, for efficiently moving data from one array (source array) to another array (target array). The move predefines require that the element type of the source and target arrays be identical. Type coercion may be used to copy arrays that have different element types.

These predefines have five parameters:

```
move_fast   (n, source, soffset, target, toffset)
move_l_to_r (n, source, soffset, target, toffset)
move_r_to_l (n, source, soffset, target, toffset)
```

These parameters are:

```
n          Number of elements to move
source     Source array
soffset    Source offset
target     Target array
toffset    Target offset
```

The source and target arrays may be the same array. The differences in these predefines stem from the assumption regarding the addresses of the source and and target arrays. Move_fast assumes that the source array address does not overlap the target array address. In other words, the programmer is not depending on the rippling of data. Move_l_to_r and move_r_to_l do not have this assumption. Move_l_to_r performs a left to right component move from the source address to the target address. Move_r_to_l performs the move from right to left.

Example

```
$standard_level 'ext_modcal'$
PROGRAM move_routines;
TYPE
   rec = RECORD
     f1 : shortint;
     f2 : shortint;
     END;
   recarrtype = ARRAY [1..5] OF rec;
   intarrtype = ARRAY [1..5] OF integer;
VAR
   recarr : recarrtype;
   intarr : intarrtype;
BEGIN
intarr[1] := 0;
move_l_to_r (4, intarr, 1, intarr, 2);
$push, type_coercion 'representation'$
move_fast (5, intarr, 1, intarrtype(recarr), 1);
$pop$
END;
```

The move_l_to_r statement uses the rippling effect to initialize the elements of intarr to 0. The following move_fast uses intarr to initialize recarr. Type coercion is used to coerce recarr to intarrtype because the elements of recarr and intarr are different. Since the element type of recarr is a record, each shortint field of the record is initialized to 0.

Building Intrinsic Files

HP Pascal provides the facility for creating, modifying and listing an intrinsic file for HPPA systems. An intrinsic file is called a SYSINTR file on HPPA systems. On a MPE/V system, the program BUILDINT.PUB.SYS is available to add intrinsic declarations to an intrinsic (SPLINTR) file.

The compiler option `buildint` is used to build or modify an intrinsic file. The file to be built or modified is specified in the string associated with the buildint option. On MPE/XL the default intrinsic file is SYSINTR.PUB.SYS if no intrinsic file name is specified. Note that, in this case, the program must have write access to SYSINTR.PUB.SYS.

Each routine declared in a program with the buildint compiler option is added to the intrinsic file. Information about each declared routine and its parameters is added, as well. If a routine with the same name already exists in the intrinsic file, the new declaration replaces the one in the intrinsic file.

Routines are declared with the EXTERNAL directive. The parameter mechanisms for extensible and default parameters may be used. The language specification on the external directive may also be used.

A program with `buildint` is similar to any other HP Pascal program except that there are only external declarations and no main body.

Only certain Pascal types may be used as intrinsic parameter types. In general, the types that may be used are limited to those which are available in most languages supported by Hewlett-Packard. These types are described in the *HP Pascal Programmer's Guide*.

Example

```
$buildint 'sysintr'$
$standard_level 'ext_modcal'$
PROGRAM build_intrinsic_file;
TYPE
   pac = PACKED ARRAY [1..1024] OF char;

PROCEDURE xxx (VAR x1 : pac; x2 : integer);
   external;

PROCEDURE yyy (ANYVAR y1 : pac; y2 : integer)
   OPTION default_parms (y1 := nil, y2 := 0)
          uncheckable_anyvar;
   external;

PROCEDURE zzz (parm1 : integer; parm2 : integer);
   external ftn77;

BEGIN
END.
```

In this example, the intrinsic file name is `sysintr` in the user's group and account. Three procedures `xxx`, `yyy` and `zzz` are added to the intrinsic file. Procedure `xxx` is a simple declaration which does not use any new system programming features. Procedure `yyy`, in contrast, uses OPTION `default_parms` and OPTION `uncheckable_anyvar`. Procedure `zzz`, according to the language directive, is a FORTRAN77 subroutine.

The contents of a SYSINTR file may be listed with the compiler option `listintr`. If no string parameter is supplied to `listintr` the contents of the SYSINTR file is output to the formal designator `paslist`.

The information in a SYSINTR file may be accessed with the `INTRINSIC` directive. Each intrinsic declaration accesses the intrinsic file specified in the `sysintr` compiler option. If the compiler option `sysintr` is not specified, the default intrinsic file is SYSINTR.PUB.SYS.

Example

```
$sysintr 'sysintr'$
PROGRAM intrinsic_calls;

VAR
  a : PACKED ARRAY [1..1024] OF char;
  i, j : integer;

PROCEDURE xxx; intrinsic;
PROCEDURE yyy; intrinsic;
PROCEDURE zzz; intrinsic;

BEGIN
xxx (a, j);
yyy (i);
zzz (i, j);
END.
```

When the INTRINSIC directive is encountered, the HP Pascal compiler accesses the information in the intrinsic file 'sysintr' and uses it for checking actual parameters and for code generation.

For procedure yyy, i is a legal actual parameter for the first parameter because it is an anyvar parameter. The length of i is not passed, however, in the call to yyy, because it is an uncheckable_anyvar parameter. The default value 0 is passed for the second parameter since it was not supplied by the caller.

In the case of procedure zzz, the compiler knows that it is a FORTRAN77 subroutine and provides reference parameters even if they were declared as value parameters in the external declaration.

Conclusion

This paper has highlighted the new systems language features available in HP Pascal. You are encouraged to try these features when writing new HP Pascal programs or enhancing current programs.

References

HP Pascal Reference Manual (31502-60005)
HP Pascal Programmer's Guide (31502-60006)

Acknowledgments

I would like to thank Jon Henderson, Ron Rasmussen, Jean Danver, and members of the Pascal project, past and present, who reviewed this paper.