# ONE SOURCE, MANY MACHINES: APPLICATION DEVELOPMENT USING HP PASCAL

**Jean H. Danver**
**Hewlett Packard Company**
**Cupertino, California**

**Overview:**

This paper presents two topics. The first is migrating a large application from MPE/V to MPE/XL in such a way that the resulting source is shared for both systems (and HP-UX as well). The second is information on migrating Pascal applications in general to MPE/XL. The HP Pascal compilers: Pascal/V, HP Pascal/XL and HP Pascal/HP-UX were originally a set of compilers for the Classic 3000 and cross compilers based on those compilers for experimental computers. This software was ported to MPE/XL and Series 800/HP-UX as native Pascal compilers. How this was done and the final result is explained. The source changes required for migration will be pointed out, as well as how they differ from what a user would have to do today. Since a compiler is an example of only one type of application, it does not have all the common migration problems. So, the compiler features supplied to aid migration of other types of applications are discussed. Migration of applications using other common subsytems such as View and Image is not covered. General tips for the migration of Pascal applications are given throughout and a list of publications that can help is included.

**The Challenge**

It was our job to produce a pair of Pascal compilers for HP-PA that was compatible across operating systems and minimized migration effort from Pascal/V. We were to do this before there was any hardware or operating systems available, and at the same time we had to provide development tools for the projects using HP Pascal. This included all of MPE/XL, SQL-based database products, most data communications software and parts of several compilers. When this started we had the Pascal/V compiler, an internal version of Pascal/V known as MODCAL and a MODCAL cross compiler for a canceled computer project known internally as Vision. The challenge was to pull all this source together and end up with a shared source system that produced three compiler products: Pascal/V, HP Pascal/XL and HP Pascal/HP-UX. The only way this could be done in the time-frame needed was to port the front end of the Pascal/V compiler.

**The Result**

Four compilers (Pascal/V, HP Pascal/XL, HP Pascal/HP-UX, and MODCAL/3000) emerge from over 250,000 lines of Pascal source code. They are targeted for two architectures and three operating systems and all share the same front end source. The source is maintained on an HP-3000 Series 68. Each compiler is created on the target machine by moving the source over a network for compilation and testing.

Logically, the source is organized in a hierarchy of directories. The main directory is called official (contains all the official source). This is actually an account called official. There are three logical directories for sources known as fe (front end), pa (precision architecture) and 3k (classic 3000). Each of these logical directories contain four other logical directories. They are proc (procedures), decl (declarations), ext, (external procedure declarations) and ob (outer block). In reality, the logical directories are represented by groups under the official account. They are named procfe, procpa, proc3k, declfe etc. The actual source files are in these groups. The fe groups contain source that is shared with all the compilers. The source in the pa and 3k groups are primarily code

generation routines that are aimed at a particular architecture. The files in the ob groups are the ones actually compiled. They include the files in the other groups during the compilation. As a rule they are set up as job files. The latest usl files used to create compilers for testing are also there. Source is managed using an internal source management tool. The actual source files are named after the procedure they contain (every procedure is in a separate file). Having all the files in one account facilitates dual development with HP-UX and the requirements of the source management system. It also makes it alot easier to replicate the compiler source somewhere else, if needed. All the includes are done without reference to account name, for example, which allows putting the source in a different account very easy. A picture of the source organization is in Figure 1.

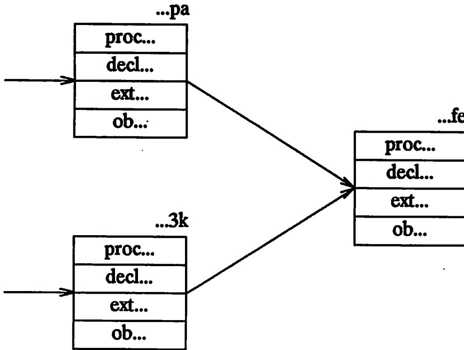


**Figure 1**

Using a Classic 3000 development environment is not the only logical choice. An MPE/XL machine could be used. Cross development for the Classic 3000 can be done by using compatibility mode. Object files between MPE/XL and the Series 800 HP-UX are compatible, so the HP-UX compiler front end could be compiled on MPE/XL and transported over to the Series 800 for linking.

Because part of testing for the Pascal compiler is to compile itself on the target machine with the target OS, source and tests have to be moved anyway. So, each compiler is routinely produced on the target machine. All the machines are connected in a network. The engineers develop on whichever one they choose. HP-UX is not suitable for base development because of the lack of compatibility mode and 3000 format floating point emulation. Though debugging and development that does not involve those features is frequently done under HP-UX because some of the engineers prefer it (especially those who learned UNIX* in school).

**How We Did It**

**Changes Required for Migration & Source Sharing**

A. Pascal Dependencies

Language features presented the least of our problems. They centered around one issue -- dependencies on the Pascal/V packing algorithm. The compiler did not have too many and

* UNIX is a trademark of Bell Laboratories.

they were pretty well isolated.

There were some debugging routines that displayed pointers values as integer values. This was done with a tagless variant record that overlayed a 16 bit integer with a pointer. The problem of pointer overlay solved itself. The type declaration Smallinteger: -32768..32767 is allocated 16 bits in Pascal/V and 32 bits in Pascal/XL, the same sizes and alignment as the respective pointers. So, we were still able to display the pointers. If we had been doing variant record pointer arithmetic tricks, changes would have been required. Heap pointers on Pascal/V are 16 bit word offsets and on HP-PA they are 32 bit byte offsets. Identical manipulation does not get the same offset changes.

HP Pascal has a feature known as structured constants. This is the ability to declare a record, array, set or string constant. The feature is implemented by building the array or record constant at compile time in a buffer area. Implementation requires moving arbitrarily sized objects on arbitrary bit boundaries to arbitrary bit boundaries. This was accomplished by creating overlays using tagless variant records. Needless to say, the layout came out differently in Pascal/XL. What we did was to create two record definitions that had the same packing in both compilers. The Pascal/V one remained unchanged. The Pascal/XL one was the same structure PACKED. ShortInt was used to obtain a 16 bit integer. Conditional compilation ($IF..$ELSE..$ENDIF) was used to put the changes in source.

It is usually not difficult to create two record structures with the same packing on each implementation. The declaration may be a little different. For example, just making a structure PACKED on XL will frequently be the same as unpacked on Pascal/V. We found that using conditional compilation to cause differences in declaration was much better than changing algorithms to manipulate things. A very useful compiler option to use in the checking out of packing is $TABLES. In both Pascal/V and Pascal/XL it will print out the structure layouts. The output can be used to verify that the layouts are the same.

This solution points out several things:

- Use the predefined type, ShortInt, to get a 16 bit integer in XL. Be sure that it is not declared in the XL program, as this declaration will override the predefined type and will likely not be 16 bits.
- Manipulating declarations to get the same packing as in Pascal/V is usually preferable to changing algorithms. It is also preferable to using $HP3000_16. Why this is the case will be explained below.
- Conditional compilation is very useful for coding small differences when the source is being shared.
- Small integers with negative ranges are allocated 32 bits on XL and 16 bits on V, so pointer overlays can share the same source, as long as the overlay is for display or comparison purposes only (and extended addresses are not being used).
- $TABLES can be used to check the layout of declarations.

B. Bug Fixes

Believe it or not, there were bugs that had to be fixed that did not show up on the Classic 3000. These bugs were uninitialized variables. Garbage on the Classic 3000 tends to be zeros. Zero frequently is fine for an initial value. On HP-PA garbage really is garbage. So, some pretty confusing bugs appeared. Even though we were aware of this, having a feature work on MPE/V and not on MPE/XL or on one version of XL and not another almost always lead us to suspect the OS. (Customers suspect the compiler). But more often than not, it was an uninitialized variable on our part.

Users run into this problem with string variables very often. They might use Strwrite on a string, for example, but forget to assign a null string to it before hand. Strwrite will update the length, if it is expanded, so everything works fine on the Classic 3000. On XL, there may be some huge number for the garbage length which will result in an abort as soon as anything is done to cause a range check. Whenever you have a program abort on a string when it doesn't abort on the Classic 3000, look for uninitialized string variables. These problems come up with several of the string routines. Strmove and StrAppend are common routines that are incorrectly used to initialize variables.

String variables were rarely our problem. We were too aware of that one. Our uninitialized variable problems tended to be fields in records that a called procedure expected to have a certain value and the calling procedure did not fill in. The zero on the stack was the correct value on the Classic 3000.

These are the random porting problems that cause the biggest headaches. My advice is to learn NM-Debug or XDB, and have faith that it is your problem and suspect uninitialized variables.

It is important also to develop programs that do not reply on undetected range errors requiring $RANGE OFF. One great advantage of Pascal is that random garbage usually results in a range error of some sort, so the problem is detected fairly soon. Do all your development and testing with $RANGE ON.

C. Operating System Dependencies

**Extra Data Segments**

The operating system dependency relied on the most, which did not map onto XL, was extra data segments. The Pascal/V compiler makes heavy use of extra data segments for reducing stack use. For example, all symbol names and constants are kept in extra data segments. Using extra data segments to save space makes no sense on HP-PA. There is lots of space. The logical thing was to switch to heap use. However, lots of code in the 3000 compiler was written with extra data segments in mind. We did not want to rewrite everything and, more importantly, we wanted to share source. To accommodate this, the extra data segment accessing routines were changed to do heap access instead.

This turned out to be trickier than first expected. It involved a mapping of a segment of 16 bit words onto a heap variable of 32 bit words. It was decided to keep all "extra data segment" addresses in terms of 16 bit offsets. This was not exactly intuitive for HP-PA, but did result in shared algorithms. All the "messy" stuff was encapsulated in a few routines with names like XSegPut and XSegGet. They handled all the heap addressing on HP-PA and extra data segment addressing on MPE/V.

In addition to space availability, efficiency considerations also contribute to differences. For example, the structured constants part of the compiler had a buffer scheme in the stack to reduce calls to the extra data segment routines. This made no sense in the HP-PA compiler. It amounted to double moving. The 'extra data segment' was heap. It was much more efficient to go there directly. This issue caused a rewrite of some of the structured constant building algorithms to make them more general. The bottom line is that programs should not only have machine dependencies isolated because you may have to port some day, but also one should isolate in the same way (or resist) those 'efficiency' algorithms based on your particular machine.

There will always be ways to make things run faster on a particular machine. You should be careful to only tune those things which need to be tuned. One guideline is to tune only those

things which are on the most frequently used path. Another is to only make machine dependent improvements only when there is a measurable performance improvement of greater than some percentage.

**Command Interpreter**

All job streams had to be recreated, just like yours, because of differences in running jobs, libraries and linking and loading programs.

The compatibility between the MPE/V CI and the MPE/XL CI permitted a phased migration of the development environment. Initially, the environment was ported intact with only necessary changes being made, e.g. using the Linkeditor versus the Segmenter.

One interesting quirk of which to beware is that new 'reserved words' have been added to the MPE/XL CI. A UDC named DO which was brought over caused considerable confusion whenever it was executed until it was realized that it is one of the CI's new 'editing' commands. *Unlike their counterparts in the MPE/V CI, these commands cannot be superseded by a UDC.* As familiarity with MPE/XL increased, the development scripts were altered to take advantage of two of the MPE/XL CI's most powerful features: **command files and environment variables.** These are not backward compatible, but can be used for compatibility mode compilation.

The interface between our command files and environment variables is similar to that of modules and imports or procedures and parameters. As many operations as possible (compiling, linking, error reporting, etc.) were modularized into command files. Since as many as fourteen varieties of the self-compilation script exist, this modularization greatly decreased the time necessary to modify any part of the compiler generation process.

Information for the command files is provided via environment variables.

Working from a basic self-compilation template, the environment variables are initialized to provide a specific compilation environment (destination of the resulting object files, optimization, symbolic debugging, etc.) Among the environment variable capabilities which we have found most useful are:

- The predefined environment variables, particularly those specifying the invoking user (hpuser) and group (hpgroup). This allows a single command file to be tailored to behave differently dependent on who is invoking it. Others which are especially useful are hpdatef, hptimef, hpusercapf, hpwaitjobs, and hpautocont.
- The ability to set environment variables to string values. This allows file names, e.g. the compiler to be used, to be passed to command files. When combined with the ability to parse and compare these strings, powerful preprocessing may be performed. For example, we were able to produce an environment that looked like the scripts we used in HP-UX. This made the operating environments look the same so that the software engineers were not having to do an environment switch in their heads all the time.

**Intrinsics**

Some intrinsics changed from MPE/V to MPE/XL. Most intrinsic changes are hidden by the intrinsic mechanism. All intrinsics used by the compilers were declared as intrinsic, so most changes were automatically taken care of. Some intrinsics did cause minor changes. These were those for message catalogs, CREATEPROCESS, and traps. The compiler calls intrinsics, such as FOPEN, that have 16 bit returns in XL. The predefine, ShortInt, was used here.

Both of these were isolated by conditional compilation. Our use of CREATEPROCESS was based on preserving stack space on the Series/V. The compiler actually creates a separate process to do the cross reference. This use was removed from the XL compiler.

The Pascal compiler tries to do some reasonable recovery if a trap occurs while it is running. The code was added to use the new set of intrinsics. The old MPE/V intrinsic, XLIBTRAP, does work. The new intrinsics, ARITRAP, HPENBLTRAP, and XARITRAP, are on HP-UX as well as MPE/XL, which fit our purposes very well for shared source with HP-UX and they give better control of trap handling in general. As a result they were used in the HP-PA compilers. They are not backward compatible with MPE/V. These routines are all documented in the *HP Pascal Programmer's Guide*.

The format for intrinsic files changed from Classic 3000 to HP-PA. This is not a concern for user programs since compilers are the only applications that access intrinsic files. Some users have their own intrinsic files. These must be converted to the new format. How to do this is explained in the *HP Pascal Programmer's Guide*. The compiler had to change the part of the compiler that accessed intrinsic files because of the new format.

Pascal/XL builds intrinsic files. In fact, the only way to build intrinsic files on XL is to use the Pascal compiler. As a result of this, the compiler group frequently became involved with the production of the system intrinsic file for XL. Using the 'wrong' or out of date system intrinsic file was a frequent source of problems.

**Architecture**

Sixteen bit arithmetic causes hardware traps on the Classic 3000 when an operation's result is greater than 16 bits. The compiler used this fact to optimize code generation. Rather than doing arithmetic in 32 bits for a binary expression whose result had to be 16 bits we used 16 bit arithmetic, allowing a hardware trap to catch range errors. This could not be done on HP-PA, since all arithmetic is done in 32 bit registers and no trap would occur. So, in the HP-PA compilers we needed to generate range checking code to catch the range error. Where range checking code is to be emitted is determined during semantics processing. During code generation we either generated checking code or not depending on the semantic result. Our range checking algorithms were dependent on the Series/V architecture. These were re-written in a more general manner and some of the processing was delayed until code generation, which is not a shared part of the compiler.

There were internal base type representations of numbers based on whether they were 16 or 32 bit arithmetic. This was determined during semantics. Here was another case where the algorithm was dependent on the Series/V architecture. There was no need for the 16 bit representation and it caused some amount of confusion, since what we did at code generation was affected by the representation of the number. We also experience some problems with I/O in the run time library which keyed off the internal representation of the number. User programs should not experience these kind of difficulties, but you never can be sure. Whenever you are doing anything that has a data dependency, there might be migration issue.

D. Special Feature Dependencies

There are some features of the Pascal compilers that turned out to be essential for source sharing. The most important of these has been mentioned several times. That is the conditional compilation mechanism. Obviously, one does not want to duplicate source for a minor change. It took some experimentation to get this feature right for large application development. $SET options which give values to the conditional variables must appear before the PROGRAM header in a compilation. All variables must be given values (no defaults) and

they cannot be changed later on. These may sound restrictive, at first, but no misspelled variables get default values and there are no behind-ones-back changes later in the compile. We did not want bugs from compiling or not compiling a piece of source by mistake.

Also essential is the nesting of $IFs. That is, to be able to put a $IF inside of a $IF:

```
$IF 'HP-PA'$
....
   $IF 'XL'$
   ....
   $ELSE$
   ....
   $ENDIF$
....
$ENDIF.
```

This may not seem so at first glance, after all, there is only the 'Classic 3000' and HP-PA. Well, what about HP-UX. At the high level we have machines (3000 and HP-PA), which may contain operating system conditions (UX and XL).

Conditional compilation also allows conditional development. For example,

```
$IF 'new_xyz_feature'$
 ....
$ENDIF$
```

The production compilation sets 'new_xyz_feature' to FALSE. When it is debugged, it is set to TRUE or the $IF was removed from the code.

We use two other features of the compiler to make self-compilation possible. We need to have source on the target machine for a self compilation test. Different operating systems have different file naming rules. This presents a problem for source files included with $INCLUDE, of which Pascal has thousands, since the OS specific file name is specified in the include. Filename.group.account is not what UNIX * expects to see. A compiler option, $Convert_MPE_Names, was developed to convert filename.group.account to ../account/group/filename. This enabled an accounting structure to be set up in HP-UX that would compile the same source. The problem does not arise between 'Classic 3000' and XL. $Convert_MPE_Names is being released in HP Pascal/HP-UX.

Another useful internal option is one that logs to a file all the files that are included in a set of compilations. The result is a list of the files that need to be moved across the network. A cross compile is done with the conditional compilation flags set for the target machine, the sole purpose of which is to get the file list. This is also being released in future compilers with a yet-to-be-determined name.

## War Stories, or Things You Don't Have to Worry About

A. The Compiler or the Operating System or the Computer

* UNIX is a trademark of Bell Laboratories

Developing an application for an operating system and a computer that did not exist certainly had its challenges. It would take a book to describe the process, the things that did not work, and the 'temporary' solutions that were required. A few things will be mentioned to give you the flavor of what went on.

A native compiler requires a native operating system on, at least, prototype hardware. There were simulators which were hopelessly slow and emulators which were in short supply and not real fast either. We were able to use HP-UX for development. Since the source was shared, work done on the compiler applied to both compilers. HP Pascal was a working native compiler on the 800 awhile before XL was ready to run it. When XL was ready, we were ready, too. This greatly speeded up XL development.

MPE/XL is written in Pascal/XL. It often was not easy to determine if the OS had a bug or the compiler had a bug. When there was a doubt, the compiler on HP-UX could sometimes be used to make the determination. That did not always work. There were many evenings spent with Pascal and MPE/XL engineers huddled together trying to determine what was going on. Both compiler and OS work might stop when this happened. The thought of all that engineering talent going to waste was pretty motivating to solving problems. NMDebug was not yet working well. Problem areas tended to be XL source management, a reliance on bugs that got fixed in a new version, uninitialized variables and too many other things to remember.

B. Rollovers

A rollover is when a change was being made, usually to generated code, that is incompatible with previously compiled code and requires every piece of code to be recompiled. That doesn't sound so hard. You just get a new compiler and recompile your source, right? Right for a compiler user, but not the compiler. It does not make the user happy and it is time consuming, but it is straightforward. Well, there are real pump-priming problems with rollovers. We did two major ones during HP-PA development. These were changing the code generation for procedure calls and changing the convention for the external names of procedures (the link names were changed from upper to lower case).

1. New External Naming

In the beginning the operating system runs in the old naming convention and expects code in the old naming convention. To prime the pump, a Pascal compiler is produced that runs in old naming and produces new naming convention. The implications of this is that the Pascal project had to produce and maintain a variety of compilers for awhile:

a. old/old - For the users that were still using the old operating system and compilers that run on them. They tended to be alpha test sites.

b. old/new - For the users that had the old operating system and needed to compile their code to run on the new.

c. new/old - This was for the operating system which expected user code to be in the new names (Pascal is a user program), but needed old names produced because that was what they were still using.

d. new/new - The end result. This was regular users on an operating system expecting new names.

All the assumptions coded-in concerning external names needed to be discovered and removed or changed before the operating system would work. What actually was done was, rather than rolling themselves, the OS modified code so that it required user code to be in new naming, but it ran in old naming. This resulted in rolling the rest of the world

faster, but compounded Pascal's version problem. Hundreds of little routines were written called stubs. A stub was a routine in one external name that turned around and called the real routine in another external name. Routines called by users and the OS linked in stubs in either direction depending on the external names of the actual routine. Once the rest of the world was running in new names the operating system switched itself over. The whole process took months. Some funny bugs resulted in the end. For example, a stub for a particular routine was written in new naming that called old naming. Later the routine was converted to new, and a stub was written expecting old and converting to new for operating system use. As a result, our code was linking in a stub that got linked to another stub that reversed the names. Everything worked, even though there was two unneeded stubs. The OS converted, deleted the old to new stub and suddenly we could not load programs anymore. It was quite a surprise, until someone realized what was going on.

2. Procedure Call

The procedure calling convention is the code sequences that are used to cause a procedure call. There has to be agreement in an operating system what these code sequences are in order for things to work. Changing the procedure calling convention was much more complicated than changing the external names. There were blocks of assembly code that were coded in the old convention that had to be recoded and debugged. Debugging each one required all the others called before it to be debugged. The problems could only be discovered serially. The process also required the same flavor of compiler versions mentioned above. The details of this rollover will not be included here. It belongs in a book like *The Soul of a New Machine*. It is the kind of change that can not be made after a product is released.

C. Getting in Your Way When Producing Yourself

Writing a compiler in itself is a bit of a chicken and egg problem. For the most part Pascal/V was used to produce an XL-compatible cross compiler which was used to produce the native compiler. When there was a more stable operating system, the native compiler was used to produce the next native compiler. However, the cross compiler is still used for include log lists, making sure everything compiles (the master source is on the Classic 3000) and debugging. As a rule there is no problem adding a feature to a compiler written in itself once you have a version of the compiler. This works because you do not need the feature to write the code to put it in. The most common problem that arose was destroying the cross compiler because of some interaction of the new feature with an existing feature that did not show up until, well, it was too late to back up. For example, someone would change a global declaration, check all their changes in and then the resulting cross compiler would not work. This left everyone else in a state where they could not build a compiler to check out their changes and, if XL wanted a hot one fixed, the project was in a hot seat.

We would get into what looked like chicken/egg problems when a run time library routine would change its interface. The compiler could not produce itself without lots of intervention. This was not difficult, just detailed, and one had a tendency to not realize it until the compiler failed to produce itself. This is how it works. New source (with the new library interface) would be compiled with the old compiler which needed the old run time library. The resulting compiler ran with the old library and produced source that required the new library. This untested compiler was delivered to the back end project to produce a back end that did the new call. It was also used to compile the source again. That created a front end that required the new routine and produced source that did it as well. It was combined with the new back end and a new library to result in the final compiler.

Later, the run time library routines were made extensible, so this type of roll would not have to

be done again. (When compiled with Option Extensible, parameters can be added to routines without affecting existing compiled programs that call the routine.)

**Problems We Did Not Have**

A. Stack Space Limitations

Pascal programs compiled on the classic 3000 are limited to one 32k byte segment for data space (heap, global data and stack) at run time. XL programs have a much, much larger limit. This problem can interfere with portability, as pointed out in the extra data segment discussion above. It also limits backwards compatibility, as well. All kinds of decisions are traded off. Structures on the Classic 3000 are usually organized to maximize space savings. Giving up a little space in a structure may increase portability, backwards compatibility, simplicity and decrease speed. But, it is not possible when you run out of space. This problem was worked around internally by using an 'extended heap'. This feature increased heap size by using a cache scheme in the stack segment and putting heap overflow into extra data segments. When extended heap is needed things can go pretty slow. However, it greatly increased source sharing between Pascal/V and Pascal/XL.

The Pascal/V compiler itself now runs in extended heap when needed. This removed most compiler limitations from getting in the way of dual development. User programs themselves cannot run in extended heap, so cannot get around this potential compatibility problem.

This feature has not been given to customers because of the performance of applications running in extended heap and the limits on what can be done with an extended heap. Heap addresses are not stack pointers anymore. Any changes of pointer values in a program will not work. TOOLSET does not know how to debug extended heap pointer values. Files cannot be put in an extended heap. However, if this feature is important to you, you should make your needs known to Hewlett Packard.

B. Existing Data

Compilers, as a rule, do not have any existing data files or data bases, so there was no data that had to be converted or exist in two environments. Features were put in the compiler to support user applications with these types of conversion problems. The section below discusses them. Compilers do read files, which could have caused some conversion of formats. However, the files we read were ASCII files, so no conversion was necessary. You should keep this in mind while doing development for dual environments. Use ASCII files or Pascal data files with data packing that has the same layout in both Classic 3000 and HP-PA.

**Run-Time Support Changes**

Along with the compiler, a run time library is provided with Pascal. In Pascal/V, the run time library handles I/O, heap support, strings, some set manipulations, and some of the predefines, such as Hex, Octal and Binary. There were substantial changes made to the run time library routines for I/O, heap and strings. Set routines were no longer needed. The code generator provided the run time support.

The I/O routines on MPE/V were written in SPL for historical reasons. These were re-written in Pascal.

The heap routines manipulated the 3000 Classic 3000 DL-DB area of the stack. This was rewritten to use HP-PA addressing instead. It also had to be changed to take into consideration HP-PA alignment restrictions for data. This was another case of an architecture dependency.

Strings had a similar problem. The resulting size of some string expressions can not be determined at compile time. This did not present a problem on the Classic 3000 because the data stack could vary in size. This was not true in HP-PA, which requires fixed size frames. Those expression values are now done in the heap.

**Compiler Support for Customer Applications**

Most of the information here is covered in the *HP Pascal Programmer's Guide* and the *HP Pascal/XL Migration Guide*, and you should use those guides for reference.

Migration was and still is an important part of XL development strategy. Very early in the development cycle there was a task force devoted to drawing up the strategy and the technology that would be developed to achieve migration. The task force recognized that migration was not something that happened overnight. Classic 3000s would be around for years to come. An application may be converted in stages. Therefore, users will need to have parts that ran on both MPE/V and MPE/XL and shared MPE/V data. What this meant for the Pascal/XL compiler was:

1. All features of Pascal/V must be in Pascal/XL
2. Pascal/XL must be able to run in an environment where the data is in Series/V format
3. Pascal/XL must provide a way to enable conversion of Pascal/V data files to Pascal/XL data files

A pair of compiler options and two conversion routines were developed to accomplish these goals. These are $HP3000_16, $HP3000_32, StrConvert and SetConvert. In order to enable programs to co-exist with Pascal/V programs and other applications running under MPE/V the $HP3000_16 options were created. When $HP3000_16 is on data is packed in the same format, when possible, as Pascal/V. This means:

1. all reals are in MPE/V real format
2. strings and sets have the Pascal/V format, which is different than the HP Pascal/XL format
3. types that do not contain files or pointers are sized and aligned the same as Pascal/V
4. all data manipulation assumes MPE/V real numbers and Pascal/V sets and strings

- Accessing Data

  So, now you can interact with Pascal/V data using $HP3000_16. The two exceptions are files and pointers. Native mode pointers are 32 bits. So, structures with pointers will not be laid out the same as in Pascal/V, nor will structures containing files. This ordinarily should not matter. Files are not assignable, so structures with files will not be stored in data files anyplace. Pointer values make no sense, except in a particular invocation of a program. Hence, they rarely get stored as data. If you have a problem here, use a 16 bit integer, such as Shortint, instead. That has the same size and alignment as the Pascal/V pointer.

  It should be pointer out that $HP3000_16 should only be used to manipulate MPE/V data. It was not designed as an alternative packing algorithm. Code generated to manipulate strings, sets and real numbers is not the same as when HP3000_16 is not in effect. You cannot mix routines compiled with HP3000_16 with ones that are not. Some rather strange things may result.

  You can write a program that can manipulate both types of real numbers (IEEE and MPE/V). However, each type must be in different procedures compiled separately. Since we discovered that the compiler could make use of this, it appears that some users programs may also have a need. A sample program is in Exhibit 2 that shows how to do it.

$HP3000_16 does not always need to be used to access Pascal/V or MPE/V data. It is only needed when the data or layout is different. It may be possible to use the same techniques that were discussed under Pascal dependencies. Declarations may be modifiable to create identical layouts. ShortInt can be used to get 16 bit integers. A simple type renaming may be all that you will need.

Data in ASCII files is the same in both environments. So are many simple structures such as integers, char, Packed and unpacked arrays of integer and char. If your external data is of this form, the same program will run in both environments.

- Converting Data

In order to convert an application with existing data, the data may have to be converted as well. To support the conversion of data, the compiler option $HP3000_32 and the routines StrConvert and SetConvert are provided. These, in conjunction, with the system intrinsic, HPFPCONVERT, are all that are needed to convert Pascal/V data files.

$HP3000_32 can only be used when $HP3000_16 is in effect. It will produce a structure that is laid out identically to the HP Pascal/XL packing. Its purpose is strictly to allow programs to be written to convert data files. Strings, sets and real numbers, for example, have the default XL packing and cannot be manipulated in the program. They are not assignment compatible with HP3000_16 strings, sets and real numbers. The conversion routines are used to obtain values for variables of these types.

Strconvert converts a Pascal/V string to a Pascal/XL string and has the form, StrConvert(PascalVstring,PascalXLstring). Setconvert will do the conversion in either direction and has the form SetConvert(VorXLSet, OtherFormatSet). So be careful with SetConvert. You could destroy your data if you get the parameters out of order.

There is a good example of the use of these options for file conversion in the *HP Pascal/XL Migration Guide*. It is repeated in Exhibit 1 with some slight improvements. As you can see, writing a program to convert a data file is quite simple, short and straightforward. The only major complication would be tagless variants. When there is an overlay with a tagless variant, you can not determine what the type of the actual data is. This makes it difficult, to say the least, to convert it.

- Switch Stubs

There are, of course, legitimate exceptions to everything. $HP3000_16 is frequently used in writing routines that call switch stubs. Switch stubs data structures are HP3000_16. Just don't use strings and, if sets have to be used, make sure the layout is the same as Pascal/XL default (this would be the case for unpacked sets that take up multiples of 32 bits in Pascal/V). Files are out of the question. The control blocks are completely different. Use Fnum, if that has to happen.

**Conclusion**

Source sharing for applications that are targeted for MPE/V and MPE/XL is quite feasible. It offers an opportunity to leverage an implementation investment of the past and the future. If the future includes HP-PA with HP-UX, the opportunity continues. The Pascal project in Hewlett Packard's Computer Language Lab is successfully doing this. The result is a more reliable, compatible compiler in a very short period of development time.

**Useful Publications**


1. *HP Pascal Programmer's Guide* (31502-90002 or 60006)
2. *HP Pascal/XL Migration Guide* (31502-90004)
3. *Introduction to MPE/XL for MPE/V Programmers* (30367-90005 or 60004)
4. *MPE/V to MPE/XL:Getting Started* (30367-90002 or 60002)
5. *Switch Programming Guide* (32650-90014 or 60030)


**Acknowledgements**


I wish to thank the members of the HP Pascal group, past and present, for ideas, memories, suggestions and editing.

**Exhibit 1**

The following program illustrates the migration of a data file from Pascal/V to HP Pascal/XL.

```
$HP3000_16$
PROGRAM Convertfile(file1,file2);

CONST
  HP3000_32bit = 1;
  IEEE_32bit = 3;
  RoundToZero = 1;

TYPE
  Arr1 = ARRAY[1..10] of -32768..32767;  { 20 bytes allocated }
  CMrec =
        RECORD
        f1:char;
        f2:Boolean;
        f3:string[40];          { 44 bytes allocated }
        f4:Arr1;
        f5:real;                {MPE/V representation; 2 byte aligned}
        f6:set of 0..15;        {2 bytes allocated }
        END;
 NMArr1 = $HP3000_32$ ARRAY[1..10] of -32768..32767; {40 bytes allocated}
 NMRec = $HP3000_32$
        RECORD
        f1:char;
        f2:Boolean;
        f3:string[40];          { 48 bytes allocated }
        f4:NMArr1;
        f5:real;               {IEEE representation; 4 byte aligned}
        f6:set of 0..15;        { bytes allocated }
        END;
 file2type = $HP3000_32$ FILE OF NMRec;

VAR
  file1: FILE OF CMRec;
  file2: file2type;
  v1: CMRec;
  v2: NMRec;
  inx: 1..10;
  status : integer;
  except: -32768..32767;

PROCEDURE hpFPConvert; Intrinsic;

BEGIN (*Program Convertfile*)
Reset(file1);
Rewrite(file2);
WHILE NOT Eof(file1) DO
  BEGIN (*Read and Write*)
  Read(file1,v1);
  WITH v1 DO
```

```
    BEGIN (*Assign the components*)
    v2.f1 := f1;
    v2.f2 := f2;
    StrConvert(f3,v2.f3);
    FOR inx := 1 TO 10 DO
      v2.f4[inx] := f4[inx];
    hpFPConvert(f5,v2.f5,HP3000_32bit,IEEE_32bit,status,
              except,RoundToZero);
    SetConvert(f6,v2.f6);
    END; (*Assign the components*);
  Write(file2,v2);
  END; (*Read and Write*)
END. (*Program Convertfile*)
```

**Exhibit 2**

{This program manipulates real numbers as IEEE and calls a routine that will manipulate them as 3000 reals. It is responsible for all the real number conversions and uses the intrinsic, HPFPCONVERT to do them. Note that this routine is compiled without HP3000_16 because we want IEEE manipulation. It must be compiled separately from the routines that do 3000 manipulation.

The option $CHECK_ACTUAL_PARM 0$ is set to get rid of a bunch of linker warnings. Parameter checking needs to be turned off because the linker knows the difference between 3000 and IEEE reals and will generate a link error}

```
$CHECK_ACTUAL_PARM 0$
PROGRAM RealHPPA(Output);

{The procedure, RealAdd, adds two real numbers as IEEE and calls a routine
that adds them as 3000 reals and passes back the result. It converts the
result to IEEE real and prints it out. Reals are converted to 3000 real
before calling the 3000 add routine.}

PROCEDURE RealAdd;

CONST
  HP3000_32bit =1;           {Parameters for calls to HPFPCONVERT}
  IEEE_32bit = 3;
  RoundToZero = 1;

VAR
  r1_IEEE,r1_3000,
  r2_IEEE,r2_3000,
  r3,
  r4_IEEE,r4_3000 : real;
  status : integer;
  except: -32768 .. 32767;

PROCEDURE Real3000Add(
  r1,r2:real;
  VAR r3:real); EXTERNAL;

PROCEDURE hpFPConvert;INTRINSIC;

BEGIN
r1_IEEE := 1.3;
r2_IEEE := 1.2;
r3 := r1_IEEE + r2_IEEE; {done in IEEE}
writeln('IEEE value: ', r3);

{Convert reals to 3000 format for call the Real3000Add)

hpFPConvert(r1_IEEE,r1_3000,IEEE_32bit,HP3000_32bit, status,except,RoundToZero);
hpFPConvert(r2_IEEE,r2_3000,IEEE_32bit,HP3000_32bit, status,except,RoundToZero);

Real3000Add(r1_3000,r2_3000,r4_3000);
```

```
{Convert result back to IEEE. All reals in this routine are treated as
IEEE. So, conversion most be done before doing anything with the result.}

hpFPConvert(r4_3000,r4_IEEE,HP3000_32bit,IEEE_32bit, status,except,RoundToZero);
writeln('Converted back value: ',r4_IEEE);
END;

BEGIN
RealAdd;
END.
```

{This is the subprogram that does the 3000 real manipulation. It assumes that all the numbers it sees are in the 3000 format. $HP3000_16 is used to accomplish this. It applies to the entire compilation unit. Hence it must be compiled separately from part of the program that does IEEE reals. NOTE: All structures are HP3000_16 and hence are incompatible with all structures compiled without HP3000_16.}

```
$HP3000_16$
$SUBPROGRAM$
PROGRAM Real3000(Output);

PROCEDURE Real3000Add( r1, r2:real;
  VAR r3:real);

BEGIN
r3 := r1+r2;
writeln('3000 real value: ',r3);
END;

BEGIN
END.
```

HP3000_16 unit is in a source file called, real1. IEEE unit is in source file called real2. The commands are as follows:

```
:pasxl real1, real1obj
  .
  .
  .
END OF COMPILE

:pasxl real1,real1obj
  .
  .
  .
END OF COMPILE

:link from=real1obj,real2obj;to=realprog;parmcheck=0
INCOMPATIBILE PACKING: output (REAL1OBJ, REAL2OBJ)  (LINKWARN 1503)

:run realprog

IEEE value: 2.50000E+00
```

3000 realvalue: 2.50000E+00
Converted back value: 2.50000E+00

END OF PROGRAM

Note that the parmcheck=0 option to the linkeditor is necessary to prevent a type incompatibility error with output. $CHECK_ACTUAL_PARM 0$ is not necessary in the RealHPPA program because of this option. A lot more warnings would be given here instead.