1 9 8 4
HP3000IUG
ANAHEIM
CONFERENCE

# PRESENTATION ABSTRACT

*Title:* Portable Development Between the HP 3000 and a Microcomputer Using Pascal.

*Author:* Kenneth C. Butler
Manager, Technology
Imacs Systems Corporation

*Abstract:* Corportations, which are now facing the appearance of microcomputers in their user community, are often discovering that many requests for microcomputer applications require custom programming. This usually means hiring contractors and accepting systems written in languages specific to one microcomputer. Supporting users with such systems becomes problematical when the in-house data processing staff has no expertise in the language chosen, and when data adminstration and auditing staffs attempt to verify the proper use of corporate assets. With Pascal/3000 it is possible to develope portable applications between the HP 3000 and microcomputers which use one of the common dialects of Pascal. Using features of Pascal/3000 to construct a carefully chosen subset of extensions found in microcomputer Pascals, programs can be written which compile on either the HP 3000 or microcomputer with few, if any, changes. It is also possible, in some microcomputer Pascals, to construct intrinsics that mimic many common MPE intrinsics. Finally, some microcomputers support environments uncannily close to a "single tasking MPE" environment; i.e. 64Kb separate code and data stacks; a hierarchical file system whiuch can simulate "groups" and "accounts"; and program and system code libraries.

1 9 8 4
HP3000IUG
ANAHEIM
CONFERENCE

# Portable Development Between the HP 3000 and a Microcomputer Using Pascal.

Kenneth C. Butler
Imacs Systems Corporation

## 1. Introduction

For many corporations the much heralded microcomputer revolution is finally arriving. One way this happens is the gradual acquisition of personal desktop computers by key departmental managers, who intended to use "primary" off the shelf applications, like spread sheet and word processing programs. As these applications prove successful, more microcomputers are purchased, are given to the professional and secretarial level persons throughout a department, and more off the shelf applications, like data base managers, come into use. Somewhere during this process requests begin to surface for applications which demand just a little bit more than the off the shelf programs can provide. These requests may involve applications such as:

- Extracting information from files and data bases from one computer (usually the HP 3000 host) and transmitting it to another computer (usually the microcomputer). Typically this is first encountered with spread sheet users needing current corporate data.

- Handling data in the unique way a corporation does business and which may not be easy to implement using a standard package. For example, a simple data base of international sales orders might require extensive currency conversion based upon the specific date of each transaction rather than the overall balance of an account.

- Installing a complex, but small capacity, data processing system for some service department. An example of this might be an inventory control system for a stock room; this system might include vendor information for reordering stock and customer information for "billing" other corporate departments for useage of the supplies.

It is at this point that the corporate data processing staff becomes involved, often for the first time. Other than ignoring the situation, which happens in a larger percentage of the cases than you might suppose, the dp staff can use several approaches to dealing with the situation:

- Let the users become programmers and develop their own applications.

- Hire contractors who are experts in the various microcomputer systems to do the development.

Both of these approaches usually mean accepting systems written in languages or data management systems which are specific to one microcomputer or family of microcomputers. It also usually means accepting a system which has no commonality with the HP environment that the corporation is using. Supporting users with such systems becomes problematical when the in-house data processing staff has no expertise in the language or systems chosen, and when data administration and auditing staffs attempt to verify the proper use of corporate assets.

What is ideally needed is a way for the corporate dp staff to become directly involved, using tools that they have ready access to in the HP environment. Features needed in these tools include:

- A common language should be found on both the HP and the microcomputers, so that a primary or "kernel" system can be developed on the HP and then transported to the microcomputers.
- A high degree of portability should be supported so that the same application can be developed for different target microcomputers.
- The tools on the microcomputers should allow extremely low-level access to the basic file system of the microcomputer and to its hardware, so that portions of an application which must be machine specific may be easily developed, kept isolated, and have a standard "interface" with the kernel application.

To meet these requirements we might try to use a highly flexible DBMS package, like dBase II, or Condor, but the problem here is no HP 3000 support for these types of microcomputer DBMS. Or we might try using COBOL, which exists in several microcomputer implementations, but the problem here is the general lack of low-level access, which becomes critical when we are dealing with screen handling and data communications. We can rule out BASIC because of the highly specific implementations found on each computer. In fact, the one common language which seems to fit best is Pascal.

This paper will examine HP Pascal/3000 and several common microcomputer Pascal dialects to identify areas of similarities and differences. The microcomputer dialects to be covered are:

- Apple /// Pascal
- Pascal/MT+ (CPM/80 version)
- IBM PC Pascal

The depth of coverage of the microcomputer dialects will vary, corresponding with my familiarity with them, and the extent that I have actually used them. In the case of Pascal/3000 and Apple /// Pascal, I have used both extensively and have "ported" programs of varying complexity between these two versions. I am not quite as familiar with Pascal/MT+, having only occasionally worked with it. My knowledge of IBM PC Pascal is currently quite light and mostly comes from examining its documentation, but I have attempted to include it because of the wide interest in this dialect.

In comparing these dialects, I will attempt to specify a few general rules for increasing portability in the following areas:

- Data types and packing
- Procedures, functions, and statements

I will also examine the several specific topics which seem to be of concern when developing a portable application:

- String handling
- File access
- Screen handling
- Heap management

Then I will briefly look at constructing intrinsics in the various Pascals to improve the fit of non-standard features. The issue of separate compilation will be addressed here.

Finally, I will look at the basic operating systems that these dialects use to look for common environmental features with MPE.

## 2. An Overview of the Pascal Dialects

Pascal was devised by Niklaus Wirth in 1968, and was primarily intended for two purposes: to teach fundamentals of programming as a systematic discipline, and to implement programs in a reliable and efficient way. However, the language as originally defined (Jensen & Wirth, 1974) lacked many features needed in a general purpose systems and applications programming tool. These missing features were in the following areas: interactive file operations (i.e. to terminals and communications lines); random access to disc files; the handling of arrays which might vary in size during program execution; and the separate compilation of program segments.

All of the Pascal dialects examined here have extended the language beyond the original Wirth definition to provide for these missing features. While there are at least two major standards organizations (ANSI and ISO) drafting standards for Pascal, and while many of the dialects to be examined are "based" upon one of the proposed standards, all of the Pascal dialects examined here reflect slightly different approaches to the same extensions. In addition, the UCSD Pascal system, which has become a machine independent "standard" of its own, has set the standard for handling strings which many other Pascals follow closely.

**HP Pascal/3000**

Developed and marketed by HP, this conforms to the proposed ANSI standard with extensions. It is the most extended of the languages covered. Enormous sets, arrays, the return of structured data from functions, are all things to be wary of.

Strings and string intrinsics have been provided which provided relatively close compatibility with the UCSD string implementation.

Low level support of the operating system is provided by MPE intrinsics in the SYSTEM library.

Separate compilation of code is supported and modules may be bound to the program using the SEGMENTER, or may be placed in a library for resolution at run time.

Code and data stacks are separate, with 64Kb available in each. Various kinds of system overhead can shrink the data stack, as with other languages on the HP 3000. Using MPE intrinsics, it is possible to allocate additional data segments and data may be moved into and out of the data segments onto the normal data stack.

**Apple /// Pascal**

This version of Pascal, like the version found on the Apple // family and to some extent on the Lisa, is based upon UCSD Pascal version 2.0., and is marketed by Apple Computer.

Low level support of the operating system is provided in two ways; through built-in UNIT I/O procedures, and through an external library of SOS I/O intrinsics.

Separate compilation is fully supported by the UNIT concept, which is slightly different than the method found in the other dialects. One major difference is the way data types and variables may be declared in a UNIT and then referenced by the program "using" the UNIT as if they had been declared globally in the program. Separately compiled units may be bound in a separate linking step (if they are "regular" units) or may be placed in a library for resolution at execution time (if they are "intrinsic" units). Program segments may be declared which overlay memory.

Code and data stacks are separately maintained. On an Apple ///, there is a full 64Kb available in each. Certain kinds of system overhead, such as file blocks, shrink the data stack, but buffer space for screen text and graphics are maintained separately from the data stack. By using SOS memory management calls, it is possible to allocate additional data segments and move data into and out of these segments onto the normal data stack.

**Pascal/MT+**

Developed and marketed by Digital Research, this conforms to the proposed ISO standard with some limitations and extensions.

This has word and byte extensions and also has many bit oriented features. Numerous extensions are designed to increase compatibility with UCSD Pascal. These include the implementation of STRINGs and string intrinsics, which are identical to UCSD, the provision for BLOCK I/O, and UCSD style HEAP management support, which requires two short user written routines.

Low level support of the native operating system is provided through various system library intrinsics. One feature unique to this dialect is the ability to code assembly language directly in the source of the Pascal program by using the INLINE option.

Separate compilation is extensively supported, but separate code segements must be bound by a separate linker program; the concept of an intrinsic library resolved at execution time is not supported. Separate code segments may overlay portions of memory.

Code and data stacks are not separated and come out of the same address space (about 56Kb on a CP/M-80 system).

### IBM Personal Computer Pascal

Developed by Micrsoft and marketed by IBM, this conforms to the proposed ISO standard with some limitations and extensions.

This has extensions for word and byte handling that are designed to work "close to the machine". Support for strings is the most different of the dialects examined. There are actually *two* types of strings; STRING, which is not variable in length, and LSTRING, which is variable in length. String intrinsic support is patterned after UCSD, but is not exactly equivalent as some intrinsics work only on LSTRING, and some on a combination of STRING and LSTRING.

Low-level access to the operating system is provided by various library modules, which require linking to the user program before execution.

Separate compilation is also supported, but separate modules must be linked prior to execution.

## 3. General Notes on Portability

In spite of the differences between the various Pascal dialects, it is relatively easy to devise a portable subset. When specifying a portable subset, a good rule of thumb might seem *If it's an extension, leave it out*, but we shall see that often the extensions to Pascal are highly compatible and the greatest differences actually occur in the fundamental implementations of the language itself. And where extensions are necessary or desirable and do not seem particularly compatible, it is often possible to develop modules in each of the dialects which perform very similarly.

### Data Types

The "standard" Pascal data types are BOOLEAN, CHAR, INTEGER, and REAL. Extensions to simple data types found in microcomputer dialects include BYTE, WORD and various ADDRESS types, and STRING. With the exception of STRING, these extended data types should be avoided. In all of the dialects considered, the CHAR data type is functionally equivalent to BYTE when used with the ORD and CHR functions for performing arithmetic.

Extended numeric data types also exist, such as LONGREAL, LONG INTEGER, and BCD REAL, and often their use cannot be avoided, as when greater precision arithmetic is needed for commercial arithmetic.

Rules:

- Avoid use of the BYTE data type. Instead, use either CHAR or a subrange of

0..255.

- Avoid use of WORD or ADDRESS data types. These are not particularly useful as numeric data types and are intended primarily for direct manipulation of memory, a technique that will be expressly forbidden!

- When using INTEGERs, create a 16-bit integer subrange in Pascal/3000 and use this to maintain compatibility with microcomputer Pascals.

- When using REALS, use the 32-bit data type in Pascal/3000. An exception to this might be when using 80-bit reals in Apple /// Pascal, or if 8087 support (80-bit real) is available on an MS-DOS machine. However, using extended precision on microcomputers usually means doing all arithmetic with procedure calls rather than with normal assignment statements.

- Limit ENUMERATED TYPES to 255 elements. This is to conform to a limitation on Pascal/MT+.

- Limit the range of SUBRANGE types to -32768..32767.

- Limit SETs to 255 elements. If you are working mostly with Apple Pascals, then this may be increased to 512 elements, but MT+ only supports 255.

- Avoid placing SETs as a component of an array of a record when using Pascal/MT+, otherwise the data stack may be consumed by unused portions of the record. Only HP and Apple Pascals allow packing to the minimum number of bits, but even here the structure will be "padded" to the next word level;

- Restrict all usage of "conformant" array types to modules which are intended to duplicate the functions of one Pascal in another dialect. An exception to this is the STRING type.

- Limit the maximum size of STRINGs to 255 characters.

- When commercial arithmetic requires more precision than INTEGER allows, use:

  LONGREAL in Pascal/3000,

  LONG INTEGER in Apple /// Pascal,

  REAL BCD numbers in Pascal/MT+.

  In IBM PC Pascal there is no support for signed numbers greater than 6 digits. However, an external module giving 8087 support may be available.

- Isolate all routines using high precision numeric types and be prepared to totally rewrite for each dialect.

## Packing

The original definition of Pascal included the concept of "packing" data into a format that required the smallest amount of actual machine storage that could represent an item of data. An example can probably best illustrate how this might be used.

For example:

<u>Packing to the bit level:</u>

```
var FOPTIONS : packed record
   DOMAIN              : 0..3;    {14:2}
   ASCII_BINARY        : 0..1;    {13:1}
   DEFAULT_DESIGNATOR  : 0..7;    {10:3}
   RECORD_FORMAT       : 0..3;    { 8:2}
   CCTL                : 0..1;    { 7:1}
   TAPE_LABEL          : 0..1;    { 6:1}
   NO_FILE_EQUATE      : 0..1;    { 5:1}
   reserved            : 0..63;   { 0:5}
   end;                           {16 bits total}
```

requires only 16 bits of storage! While a construct like the example given is mostly useful for interfacing to a native operating system, no doubt other uses for this will also seem desirable. Unfortunately this is one area of greatest weakness in most microcomputer Pascals. You may declare the above in HP or Apple Pascals and produce the desired effect, but not in IBM PC or MT+ Pascals. However, IBM PC and MT+ partially make up for this by supporting bit level manipulation either directly, in the case of MT+ with its bit-oriented intrinsics, or indirectly through SETS (i.e., var BITS : set of 0..15;). In addition, MT+ and IBM PC Pascals will automatically allocate storage for CHAR arrays and integer subranges to the minimum number of bytes required, rather than to the nearest word.

All Pascals allow you to *declare* PACKED variables. However only HP and Apple Pascals actually perform packing.

Rules:

- Avoid defined records designed to pack components into a single word or byte.
- If you must use such structures, try to isolate their usage into a few easily modifiable procedures.
- Do declared structures and arrays as PACKED; even when one version of Pascal may treat the declaration as a comment, other versions, which support packing, may require the declaration to perform properly.

**Procedures and Functions**

Most Pascals place little restriction on the size of a procedure or function, but Apple Pascal limits the maximum size of a procedure to about 1200 16-bit words of p-code. In practice this may be several pages of source code, so this is not as great a restriction as it may seem. There are also limitations on the number of procedures and functions that may be declared in a single compiled module; this number may vary from 127 to 255 depending upon the release number of the Pascal.

One of the more interesting features of Pascal/3000 is the ability to return a structured data type, like an array or a record, from a function. Unfortunately, none of the microcomputer Pascals examined here support this, and this feature should never be used!

Rules:

- Keep PROCEDUREs and FUNCTIONs reasonably small; a limit of two to three pages of source code should keep you within most limits.
- Keep the number of PROCEDUREs and FUNCTIONs within each separate compilation unit to less than 128.
- When using Pascal/3000, do not declare functions which return structured data types.

**Statements**

It is possible, in all of the Pascal dialects discussed here, to obtain the actual memory location of declared variables. In some Pascals it is also possible to perform arithmetic directly on pointer types. Using such techniques should be highly restricted, at best, if not totally forbidden.

Various extensions are available in some Pascals for use with boolean expressions. For example, in many Pascals all parts of a boolean expression are evaluated, even after the "outcome" has been presumably determined.

For example:

```
function INCR (A, B: integer) : integer;
   begin
   INCR := A + B;
   end;
...
...
if (AMOUNT > 1000) and (INCR(AMOUNT,100) < 5000)
```

will *always* perform the call to "INCR". Some programs, use this "side effect" to advantage, but extended boolean operators like "&" for "AND" and "|" for "OR" may terminate the evaluation of an expression before the side effect is invoked: this effect is similar to invoking the "PARTIAL_EVAL" compiler option in Pascal/3000. These operators should be avoided.

NOTE: When using IBM PC Pascal *no assumptions should be made as to whether all parts of a boolean expression are evaluated*: sometimes all parts of an expression may be evaluated or sometimes the optimization process may cause evaluation of an operand to be skipped. For this reason, if IBM PC Pascal is to be one of the targetted versions of an application, you should definitely avoid creating expressions designed to use a side effect.

Some Pascals (such as IBM PC Pascal) support the CYCLE and BREAK statements, which were "imported" from the C language, and provide extended control of FOR and WHILE loops. These should be avoided. In Pascals which do not have these statements, CYCLE and BREAK may be converted as follows:

Example:

<u>Simulating BREAK and CYCLE with GOTO:</u>

```
FOR I := 1 TO N DO
   BEGIN
   IF MONTH[I].AMOUNT = 0
   THEN {CYCLE} GOTO 1 {the next iteration of the loop}
   ELSE
      BEGIN
      TOTAL := TOTAL + MONTH[I].AMOUNT;
      IF TOTAL >= LIMIT
      THEN {BREAK} GOTO 2; {the next statement}
      END;
1: END;                {THE NEXT ITERATION OF THE LOOP}
2: ...                 {THE NEXT STATEMENT}
```

In the FOR statement, limitations should be placed on the control-variable, even if a particular Pascal dialect does not require it. The control-variable:

- should be an ordinal type.
- should not be a component of a structure.
- should be locally declared at the same level as the FOR statement which uses it.
- should not be a reference parameter to a procedure or function.

Note that while Apple /// and IBM PC Pascals are relaxed about this, Pascal/3000 and Pascal/MT+ are not.

In the CASE statement, limitations should be placed on the case selector variable, and the case constants:

- the case selector should be an ordinal type (CHAR is also acceptable).
- the case selector should not be a component of a packed structure (an element from a PACKED ARRAY OF CHAR is usually safe).
- case constants should not be specified as a subrange, i.e., "'A'..'Z'".
- the range expressed by the smallest to the largest case constant should not be excessive, as many Pascals build a "jump table" of case constants as part of the object code.

Rules:

- Do not use pointer arithmetic and tricky pointer types designed to allow direct manipulation of memory.
- Avoid extensions to boolean operators.

- Avoid reliance upon "side-effects" from the evaluation of boolean expressions.
- Avoid statements "imported" from another language, like BREAK and CYCLE.
- When using the FOR statement, limit the scope of the control variable.
- When using the CASE statement, limit the complexity of the selector variable and limit the range implied by the case constants.
- When using the OTHERWISE part of a CASE statement, some Pascals may use "OTHERWISE" or "ELSE", but a few Pascals, such as Apple // Pascal have no equivalent.

## 4. A Few Areas of Specific Concern

In practice, I have found the following areas to be the most difficult when programming for portability. This is because, unlike the general implementation guidelines which can be governed by limiting or omitting specific features, the following areas usually can not be simply omitted and specific changes must be made when moving from one Pascal to another. In some cases, the best solution will be to develop custom library routines for each dialect.

### String Handling

Strings are, in effect, variable length packed arrays of CHAR. This data type is most useful for handling terminal inputs and outputs, but is virtually required when dealing with TEXT files, where the length of an input or output line will vary in length. In practice, you will find that extensions for STRINGs are the most compatible extensions among the various Pascals.

Because HP Pascal/3000 allows the return from a function to be a structured data type, like STRING, it is easier to prepare routines in Pascal/3000 to perform the same string functions as in microcomputer Pascals.

Converting numerical data to string (or character) format is highly non-standard, usually involving "writing" a variable into a string. No Pascal contains editing features similar to the PICTURE clause of COBOL. One of the first custom modules that should probably be written for each Pascal dialect is an "EDIT" package to handle this.

Rules:

- Attempt to conform to the basic UCSD string intrinsics. In Pascal/3000 it will be easiest to prepare a module that is syntactically the same as these intrinsics. Note that HP string functions are a superset of all the others.
- In IBM PC Pascal, be aware that there are *two* types of strings, a distinction that does not exist in the others.
- Develop a module with a standard interface to handle "editing" of numerical data into strings, and write a version for each Pascal dialect.

Micro-equivalents to HP Pascal/3000 string intrinsics:

| | |
|---|---|
| `setstrlen` | - no microcomputer equivalent |
| `str` | - will become COPY |
| `strappend` | - similar to CONCAT |
| `strdelete` | - will become DELETE |
| `strinsert` | - will become INSERT |
| `strlen` | - will become LENGTH |
| `strltrim` | - no microcomputer equivalent |
| `strmax` | - similar to SIZEOF |
| `strmove` | - similar to COPY used within INSERT |
| `strpos` | - will become POS |
| `strread` | - no microcomputer equivalent |
| `strrpt` | - no exact equivalent; FILLCHAR might be used |
| `strrtrim` | - no microcomputer equivalent |
| `strwrite` | - no exact equivalent; STR might be used |

## File Access

Pascal/3000 provides for the most varied (and sometimes most useful) extensions to file access over the orignal Wirth definition. In most cases there are equivalent ways of doing the same things in microcomputer Pascals, but occasionally there will be a feature that will be difficult to duplicate, or should be avoided altogether.

Opening files:

All of the Pascals allow actual file designators (i.e., file names) to be associated with a formal Pascal file designator, usually as part of an "open" procedure; i.e. RESET, REWRITE, and (sometimes) OPEN. Normally the "open" procedure which is used on a file will indicate the type of access desired for that file; i.e., read, write, or read-write (which is usually equivalent to "direct" or random access). In Pascal/3000 this is definitely the case, but in microcomputer Pascals, this is not exactly an accurate way of looking at things; rather, the type of open procedure really determines whether a file already exists (RESET) or if the file is new (REWRITE). This last point will be elaborated in the section under "Temporary files".

Examples:

Opening a file for direct access:

```
open (MYFILE, "data.ptest.dev");      {Pascal/3000}
reset (MYFILE, "dev/ptest/data");     {Apple /// Pascal}
assign (MYFILE, "B:data");            {Pascal/MT+}
reset (MYFILE);
```

Direct access:

The direct access to files is an extension that is nearly the same in all of the Pascals examined. Access is always by record number, and record numbers always start with zero. The only real concern here is that Pascal/3000 has extended the READ and WRITE procedure to permit them to be used with any type of file, including direct access. Normally READ and WRITE can only be used on TEXT files, and GET and PUT must be used on all other file types. When using Pascal/3000 you should avoid the extended usage of READ and WRITE.

Examples:

Locating and reading a direct access file:

```
seek (MYFILE, RECNO);            {Pascal/3000}
get (MYFILE);
seek (MYFILE, RECNO);            {Apple /// Pascal}
get (MYFILE);
seekread (MYFILE, RECNO);        {Pascal/MT+}
```

Appending to a file:

One would imagine that appending to an exsiting file would be a basic operation included in all Pascals. This turns out not to be the case. Only Pascal/3000 supports an append access mode (by opening a file with APPEND). In other Pascals it may be necessary to open the file (with RESET) and read through the file until EOF is reached, or even to open *two* files (one with RESET and one with REWRITE) and copy from one into the other until EOF; then you may start appending to the second file. Historically, this situation came about because the microcomputer operating systems allocated disk space on a contiguous basis, and an append operation could conceivably overwrite the physical file following the one you are appending to.

Sensing end of file:

Normally sensing end of file is not a problem, but in Pascal/MT+ if a "typed" non-TEXT file is being read then you cannot rely on the standard EOF function. This is because end of file information in MT+ is based on the number of sectors used by a file, not the exact number of bytes. In Pascal/MT+ it may be necessary to use a special end-of-file record.

Closing files:

Closing a file seems straightforward enough, but there are a few considerations to be taken into account. There is the issue of saving a file, or deleting it. Also, most Pascals permit you to "reopen" a file with RESET, which closes the file and then reopens it repositioned to the beginning. Various close options are most important when dealing with temporary files, a topic which will be covered next.

Examples:

Closing a file and deleting it:

```
close (MYFILE, purge);           {Pascal/3000}
close (MYFILE, purge);           {Apple /// Pascal}
close (MYFILE, IORESULT);        {Pascal/MT+}
purge (MYFILE);
```

Temporary files:

All of the Pascals support some form of a "job temporary" file. Normally temporary files are created by opening a file which does not exist with REWRITE, but there are minor variations to this in each Pascal.

In Pascal/3000 a temporary file may be opened by omitting the filename in the REWRITE statement, but in this case the file can not be saved. If a filename is given, then either the file should not exist, or there should be a file equate issued before executing the program making the new file TEMP. Here the temporary file may be saved by using the SAVE option with the close statement.

In Apple /// Pascal, a file opened with REWRITE is *always* temporary, preserving the contents of any existing file. Here, the file must be closed with the LOCK option to make it permanent and delete the old file.

In Pascal/MT+, temporary files are opened with REWRITE with the file name ommitted. The file is assigned a special "system" name with a numeric suffix. Closing the file will not delete it, so presumably it may be renamed after the program terminates.

Rules:

- Be prepared to customize all opening and closing of files.
- Be prepared to customize direct access of files, but don't be afraid to fully utilize this.
- When using Pascal/3000, avoid using the extended versions of READ and WRITE which permit these to be used on non-TEXT type files. Instead use GET and PUT.
- Avoid extensive use of IORESULT. If you find life easier by checking the I/O status, define a series of constants, which can be changed to suit each particular operating system, and compare IORESULT against these.
- When using "typed" non-TEXT files in Pascal/MT+, do not rely on the EOF function; instead use a special record to mark the end of the file.
- If you wish to append to an existing file, you will have to develop a custom module for each of the microcomputer dialects. In some cases this may require the reading of a file until EOF becomes true.
- If you must use low level access of disk files, develop a module with a common interface, but be prepared to write a specific implementation for each computer. A possible exception to this might be BLOCK I/O in MT+ and Apple Pascal.
- If you require keyed access, the only Pascals that attempt this are Pascal/3000 and Apple /// Pascal and then only with external intrinsic support. No doubt similar packages might be found for PC and MT+.

## Micro-equivalents to HP Pascal/3000 I/O intrinsics:

| | |
|---|---|
| append | - no equivalent |
| close | - same, with some variations |
| eof | - same; may vary with non-TEXT file types |
| eoln | - same |
| fnum | - no equivalent |
| get | - same |
| linepos | - no equivalent |
| maxpos | - no equivalent |
| open | - will become RESET |
| overprint | - no equivalent |
| page | - same |
| position | - no equivalent |
| prompt | - become a simple WRITE; |
| put | - same |
| read | - used for TEXT files only |
| readdir | - no exact equivalent |
| readln | - used for TEXT files only |
| reset | - same, with some variations |
| rewrite | - same, with some variations |
| seek | - same; may become SEEKREAD or SEEKWRITE |
| write | - used for TEXT files only |
| writedir | - no exact equivalent |
| writeln | - used for TEXT files only |

## Screen Handling

Normal Pascal I/O was not designed for use with interactive terminals. All Pascals here have circumvented certain problems by defining special file types for interactive type devices. Normally this is transparent to the programmer if the standard files INPUT and OUTPUT are used. However, no provision has generally been made for anything more sophisticated than reading and writing variables in serial fashion. For most applications, this "character mode" approach will work quite well.

However, if you wish to design "screen mode" interaction you will have to resort to programming a custom module for each associated "terminal type" for each Pascal. This module should use the same procedure calls in each version, and may include functions like:

| | |
|---|---|
| Home_Cursor | - home cursor to screen upper left |
| Clear_EOS | - blank to end of screen |
| Clear_EOL | - blank to end of current line |
| Goto_XY (Col, Row) | - move cursor to Row, Col |
| Insert_Char | - insert a blank at the cursor |
| Insert_Line | - insert a blank line at the cursor |
| Delete_Char | - delete the character at the cursor |
| Delete_Line | - delete the line the cursor is on |

and so forth...

A good model for such a package may be found in the UCSD Screen_Ops unit, however most users will have no trouble designing a reasonable list for themselves. There are really only two considerations. First, whatever you devise, use it consistently, otherwise it will loose its value as an internal standard. Second, be wary of using techniques that interact at the character (or keystroke) level. While character based interaction will work effectively on all microcomputers, it will not work particularly well on the HP 3000. This is because the HP 3000 is designed to normally pass input from a terminal read upon the receipt of a carriage return, and while this may be over ridden by an FCONTROL option, character by character on even a lightly loaded system is normally slower than most people can type.

Rules:

- Whenever possible, use normal Pascal I/O for screen interaction. While this will limit you to line and prompt oriented screens, this will always be the most portable solution.

- Avoid character based interaction. While this may work well on microcomputers, MPE does not support this without frequent and annoying delays between the typing of a character and its appearance to the requesting program.

- Keep the screen oriented routines isolated so they can be customized for different terminal or console types. On HP computers (the 3000, 120, and 150) one can probably assume that an HP terminal is being used. On the Apple /// use the .CONSOLE driver. On the IBM PC running PC-DOS 2.0 or higher, use the ANSI.SYS driver.

## Heap Management

Sometimes it is advantageous to allocate work space dynamically while a program is executing. This may be used for handling a previously unknown number of variables, or used perhaps because this feature allows for the elegant handling of "linked lists" and other complicated structures. All Pascals support this: each new occurrance of a variable is created using the NEW procedure, which allocates space for it on the "HEAP", a specially reserved area of the data stack. Items added to the HEAP can only be accessed through pointers, and normally the programmer has no knowledge of exactly where in memory a variable might reside.

Many programs can function if simply given the ability to create items when needed. Problems relating to portablility occur if a program also must periodically remove items from memory to free space. Wirth Pascal defined DISPOSE as the procedure which removes an item from the HEAP. However, this presents a problem when space must be compacted from the holes left in memeory by disposed items. Two strategies have emerged for handling this. Some Pascals perform "garbage collection" on the heap when NEW requires it. Other Pascals, such as Apple Pascals, require the programmer to perform this chore, by MARKing a position in the HEAP, then using RELEASE to shrink the stack to a prior state. The two strategies are generally incompatible. Fortunately Pascal/3000 supports them both. Pascal/MT+ also will support both methods, but IBM PC Pascal only supports NEW and DISPOSE.

Rules:

- NEW is standardly implemented. However, not all Pascals support tagged variants.
- Getting things off the heap is not standard. While DISPOSE is in HP, PC, and MT+ Pascals, Apple /// does not support this feature. While MARK and RELEASE is in HP and Apple Pascals and can be used in MT+ with a short user routine, IBM PC Pascal cannot support this feature.

## 5. Constructing Common Intrinsics

All of the Pascals discussed here allow program segements to be compiled separately from a main program, and included using a separate "linkage" program. In fact, virtually all microcomputer Pascals have implemented various "standard" features of Pascal as library modules, which must be specifically included within a program if the features are to be used.

However, Pascal/3000 and Apple /// Pascal also allow the separate compilation of "intrinsic" procedures, which may be placed in library code files and bound to a program when the program is executed.

While a detail discussion of how this is done in each of the Pascal dialects is beyond the scope of this paper, a few examples using Apple /// Pascal and SOS intrinsics can illustrate the general concept.

Many MPE intrinsics have similar SOS counterparts; such as:

| | |
|---|---|
| FOPEN | SOS_Open |
| FGETINFO | SOS_Get_Info |
| FREAD | SOS_Read |
| FREADDIR | SOS_Set_Mark followed by SOS_Read |
| FPOINT | SOS_Set_Mark |
| FWRITE | SOS_Write |
| FWRITEDIR | SOS_Set_Mark followed by SOS_Write |
| FCLOSE | SOS_Close |
| GETDSEG | SOS_Request_SEG |
| FREEDSEG | SOS_Rel_Seg |

While the parameter lists vary extensively between MPE and SOS intrinsics, it is entirely feasible to construct a "HPFILES" UNIT in Apple /// Pascals which looks and performs similar to MPE intrinsics. One may wish do this because using SOS intrinsics directly in Apple /// Pascal is faster and uses less stack space than using the built-in Pascal I/O functions, and it also is easier to duplicate Pascal/3000 features like APPEND access to a file if one resorts to SOS file intrinsics. No doubt a close familiarity with similar low-level support in Pascal/MT+ and IBM PC Pascal could produce similar results.

## 6. Microcomputer Environmental Similarities to MPE

Some microcomputers support environments uncannily close to a "single tasking

MPE" environment; i.e. 64Kb separate code and data stacks; a hierarchical file system which can simulate "groups" and "accounts"; and program and system code libraries.

Program data and code stack sizes in Apple /// Pascal and IBM PC Pascal meet or exceed those available under MPE. For Apple // and CP/M-80 systems, expect less than half the amount available under MPE. However, both of these Pascals have operating systems comming in common useage which support at least 128Kb (Pro-DOS and CP/M+), so expect about three quarters of the space.

The file systems with greatest compatibility with MPE are MS-DOS (version 2.0 and higher) and Apple /// SOS (and soon, Pro-DOS on the Apple //e). Unlike MPE, which supports only two levels of file directories, MS-DOS and SOS support nearly unlimited directory levels. (There is a practical limit of the length of a file pathname which may be specified when a file is opened.)

The ability to combine separately compiled routines is available on the HP and in all of the microcomputer Pascals covered here. The approach used is virtually the same; a program like the segmenter is available to combine modules into a new runnable code file.

The library facilities closest to MPE are found only in Apple Pascals, and in the UCSD p-System. Unlike MPE, which supports three levels of libraries, system, account, and group, Apple Pascals support only two levels, system and program. However the program library may be a list of library file names, which means that an Apple program library can effectively be up to five separate library files, which may reside under any accessable directory structure.

## 7. Conclusion

Depending upon the complexity of an application, the same kernel program can be implemented with little change on an HP 3000 and a microcomputer. However, if the same program must run on several different microcomputers then some modification for each microcomputer will be required (perhaps as much as one or two months).

The key point one should have formed from this paper is that one must adopt a long range strategy so that once a basic set of restrictions, techiniques, and modules are specified and developed for each microcomputer, they may be reused in future applications.

As a final comment, here is my ratings of the degree of compatibility between the Pascal dialects covered here, from highest to lowest:

- HP Pascal/3000 to Apple Pascal
- HP Pascal/3000 to Pascal/MT+
- HP Pascal/3000 to IBM PC Pascal
- Pascal/MT+ to IBM PC Pascal
- Apple Pascal to Pascal/MT+
- Apple Pascal to IBM PC Pascal

## BIBLIOGRAPHY

"Pascal User Manual and Report", second edition,
Kathleen Jensen and Niklaus Wirth,
© 1974 by Springer-Verlag

"Pascal/3000 reference manual",
© 1981 by Hewlitt-Packard Company

"MPE Intrinsics reference manual",
© 1981 by Hewlitt-Packard Company

"KSAM/3000 reference manual",
© 1981 by Hewlitt-Packard Company

"IBM PC Computer Language Series Pascal Compiler"†,
© 1981 by IBM,

"Pascal/MT+, language reference manual"†, release 5
© 1981 by Digital Research

"SpeedProgramming Package User's Guide"†, release 5.2
© 1982 by Digital Research

"Apple /// Pascal Programmer's Manual", volumes 1 & 2,
© 1981 by Apple Computer

"Apple /// Pascal Technical Reference Manual",
© 1983 by Apple Computer

"SOS Reference Manual", volumes 1 & 2,
© 1982 by Apple Computer

"RPS Programmer's Manual"†,
© 1983 by Apple Computer

"The Pascal Handbook",
Jacque Tiberghien,
© 1981 by SYBEX Inc.

† Manual not available separately; requires the purchase of a software product.

1 9 8 4
HP3000IUG
ANAHEIM
CONFERENCE

## AUTHOR BIOGRAPHY

*Author:* Kenneth C. Butler
Manager, Technology
Imacs Systems Corporation

*Title:* Portable Development Between the HP 3000 and a Microcomputer Using Pascal.

*Biography:* Kenneth C. Butler is the Manager of Technology for the Imacs Systems Corporation, where he is leading a research and developement program involving the use of microcomputers. He received his bachelor's degree in psychology from Michigan State University in 1968. After a tour of duty in the Air Force, he has worked in the field of data processing since 1972, starting as an IBM Assembly language programmer in a downtown Los Angeles service bureau. His first exposure to an on-line system was in 1977, when he became the system manager of a Tandem Non-Stop system. Between 1980 and 1983, he was employeed by the Twentieth Century-Fox Film Corporation, where he was the system manager for four HP 3000's and acted as the primary technical applications specialist for the TCF Branch On-Line System, one of the first large systems to use the Transact programming language. He became actively involved in office automation and the corporate use of microcomputers while at Twentieth Century-Fox, and owns his own personal computer.