Avoiding pitfalls in multi-language programming

» Return to original page

by

James Overman
Sue Meloy
Walter Murray
Jim Scaccia
Don Jenkins

Language Team members in the Software Technology Lab of the Support Technology Center of the Worldwide Customer Support Organization of:

Hewlett-Packard Company
8000 Foothills Boulevard
Roseville, CA 95747-5613
(916) 785-5672

This paper is a review of the features involved in programs written using multiple languages in the HP3000 MPE/iX native mode environment. The languages to be discussed and the short identifiers that are used to refer to the languages in the paper are:

HP Business Basic/iX Basic

HP C/iX C

HP COBOL II/iX Cobol

HP FORTRAN 77/iX Fortran

HP Pascal/iX Pascal

HP RPG/iX RPG

HP TRANSACT/iX Transact

This paper is divided into topic areas and each topic includes a list of *Pitfalls* and *Recommendations* before a more detailed discussion of the specific items and language features. The topics are:

Data Representation

Procedure Invocations

Compilation Options

Linkedit Commands

Exception Handling

Run-time Factors

Heap Management

Debugging Tips

POSIX Environment

This paper will not address the issues of interfacing with the older compatibility mode languages and data types.

That it is possible at all to intermix routines from multiple languages is due to the HP Precision Architecture Procedure Calling Convention as described in the Procedure Calling Conventions Manual, which is listed in the Bibliography.

To permit this paper to address the diverse issues and multiple languages herein, it has been necessary to generalize statements and to omit minor exceptions that might contradict those generalities. The references are the most complete description of the topics discussed and should be used as the authoritative source for actual implementation. We intend that this paper will be of use as an overview of the subject.

Data Representation

Pitfalls:

Pascal data types not generally supported by other languages include enumerated types, sets, and pointers.

Fortran REAL*16 datatype is not supported by other languages.

Cobol does not support floating-point data types.

Basic DECIMAL format is different from Cobol PACKED-DECIMAL.

RPG only passes character and packed decimal parameters.

TRANSACT supports several extended precision datatype sizes not available elsewhere.

Strings are stored in differing formats by the various languages. Only the simplest character arrays are transferable among most languages.

Recommendations:

Use the simple data types (integers, characters, and reals).

Details:

Table 1

Data Representation Equivalences Across Languages

Data Types that are highly compatible across many languages:

Pascal Fortran Cobol C Basic Transact Note ----------

Char CHARACTER PIC X char x$ X(1)

Integer INTEGER*4 S9(9)COMP int INTEGER I(9)

Longint NA S9(18)COMP NA NA I(18)

Longreal REAL*8 NA double REAL R(7) #1

PAC of n CHARACTER*n PIC X(n) char [n] x$ X(n) #2

String CHARACTER*(*) NA char [n] x$ X(n) #2

Real REAL*4 NA float SHORT REAL R(6) #1

Shortint INTEGER*2 S9(4) COMP short SHORT INTEGER I(4)

NOTES:

#1 Transact also has a type E (Real Scientific Notation) that is the same as REAL in storage. Fortran has a REAL*16 that has no other language equivalents. In Cobol, you can use the Compiler Library routines PEXTIN and HPINEXT to convert real data types to/from data types that Cobol can manipulate.

#2 Cobol and Transact support other character data types such as numeric-only and uppercase characters. RPG also supports the character datatype. Strings are stored in different formats that are generally incompatible across languages.

Table 2

Data Representation Equivalences Across Languages

Not so compatible data types across languages with best equivalents:

(~ denotes size equivalency but not identical computationally)

Pascal Fortran Cobol C Basic Transact Note ----------

Bit16 ~INTEGER*2 9(4) COMP unsigned short ~SHORT INTEGER K(4) #3

Bit32 ~INTEGER*4 9(9) COMP unsigned int ~INTEGER K(9) #3

Bit52 NA 9(18) COMP NA NA K(18) #3

Boolean LOGICAL*1 NA ~character NA NA

~Record COMPLEX NA ~struct ~SHORT REAL ~R(6) #4

Enum NA NA enum NA NA #5

Pointer NA NA pointer NA NA

Record RECORD record struct ~x$ NA #6

Set NA NA ~struct NA NA

NA NA COMP-3 NA NA P #7

Subranges

0..255 ~BYTE NA unsigned char NA NA

0..65535 ~INTEGER*2 ~COMP 9(4) unsigned short ~SHORT INTEGER K(4) #3

NOTES:

#3 Integers may hold unsigned integer values with the first bit being interpreted as a sign rather than the larger unsigned magnitudes. Transact has 12-byte integer types with no equivalents in other languages.

#4 COMPLEX storage equivalent is two real numbers in a record/structure or two-element array. Only Fortran supports COMPLEX arithmetic.

#5 C enum is always 4 bytes in size, Pascal enum can be 1,2,or 4 bytes.

#6 Basic uses PACK and UNPACK to create and decompose structures/records to and from string variables.

#7 Packed decimal data type is common to Cobol, RPG, and Transact. Basic's DECIMAL data type is not compatible with other languages. Zoned decimal data type is also common to Cobol and Transact.

Procedure Invocations

Pitfalls:

Procedure name case should be considered carefully as some languages upshift, downshift, or are case sensitive.

Procedure names which contain special characters (like underscore, hyphen, apostrophe) will frequently require ALIASing to allow access from other languages.

Parameters may be passed by value, by reference, or by descriptor, and the mode of parameter passing differs by language.

Parameters other than the simple data types may be difficult to construct and/or process in the various languages.

Data alignment can be a problem; use SYNC in Cobol, avoid CRUNCHED in Pascal.

Hidden parameters can cause interfacing problems unless they are properly accounted for in the procedure call.

Global and Common data may not be visible to routines written in other languages.

Recommendations:

Procedure names should be less than fifteen characters in length, lowercase, with no special characters.

Use simple data types whenever possible.

Avoid passing structures and arrays. Single dimension arrays of simple data types may be passed by reference.

Use parameters to pass shared data rather than global data items.

Extensible parameter lists should be avoided, when possible.

System Intrinsics (as opposed to language intrinsics) should always be externally defined via the automatic language specification that uses the SYSINTR.PUB.SYS intrinsic specification file.

Details:

Procedure names (except intrinsics) are downshifted by default for all languages except C (which uses mixed-case names). In C, you must write external names exactly as they are to be seen by the linkeditor or loader. For example, the Pascal procedure name 'ProcessNextRecord' would be downshifted by the compiler. To call it from C, you would have to write it in lower case 'processnextrecord'. By default, Cobol external names have hyphens converted to underscores. This convention can be defeated by preceding the routine name in a Cobol CALL statement by a backslash. In Pascal and Fortran, the ALIAS option may be used to provide access to names that are invalid within the language (eg. names with apostrophes). The LITERAL_ALIAS and UPPERCASE compiler options may be used to reference mixed-case and uppercase routine names.

The default external routine name in Basic can be overridden by providing an alias on the EXTERNAL or GLOBAL EXTERNAL statement. It is not possible in C to create or to call a function with a special character in its name, except for the underscore. RPG uses the EXIT operator to invoke external routines with up to a six character routine name placed in the factor 2 field - columns 33-42, left-adjusted.

Parameters may be passed by value, by reference, or by descriptor. C passes parameters by value. This is part of the simplicity of design of the C language. Passing by reference involves dealing with pointers, and the C programmer is expected to code pointer operations explicitly. To simulate passing a parameter by reference, explicitly pass the address of the desired object. A Cobol program normally passes parameters by reference, although an HP extension does allow passing by value. A Cobol routine expects its parameters to be passed by reference. Transact uses the "%" option to pass a given parameter by byte address; "#" to pass a parameter by value; "&" to indicate the parameter is a function value return. Fortran passes parameters by reference unless told otherwise. Pointers from another language may be

passed by value to Fortran which can treat them as addresses of reference parameters or simply as identifiers to be passed to other languages' routines. Passing function references as parameters is a complex area and should be avoided unless required for traps and sorting.

In Basic and Pascal, the EXTERNAL directive allows for the specification of the language in which an external procedure was written. The language specifier will cause certain language specific adjustments to the procedure calling code to be made, thus facilitating such procedure calls. For example, PASCAL will convert its string format to that of C during a procedure call, and will convert the string back to the Pascal format upon return, thereby updating the string length. In Basic, for BYTE String$ parameters, the address passed is a pointer to the start of the string data. However for String$, if the language specifier is BASIC, the address passed is a pointer to the maximum length. For a non-Basic specifier, the address passed is a pointer to the current length, and the value of the maximum length is passed as a hidden parameter immediately following the string address. For Basic, if a called routine written in another language changes the length of a string parameter, the current length of the string must be updated. This can be done by the called routine or by the SETLEN statement, or automatically (for the Pascal specifier) after returning to the Basic routine. In Fortran, the ALIAS directive includes the external language specifier, and allows specification of individual parameter passing modes of value, reference, and descriptor.

Arrays are normally passed by reference. Within Basic the address passed will be a pointer to the start of the array information block. Otherwise, for other languages the address passed will be a pointer to the start of the array data.

Arrays of greater than one dimension are frequently a source of parameter passing problems. Fortran stores arrays in column-major order, while other languages use row-major order. When passed as reference parameters, the compilers often add hidden parameters that contain the maximum indexes

and other information. Fortran uses hidden parameters for assumed-size arrays and for extensible parameters lists. Pascal uses hidden parameters for ANYVAR, generic strings, multi-dimensional arrays, and extensible parameter lists. In Pascal, use of the language specifier FTN77 on the EXTERNAL directive will automatically provide the hidden array size for Fortran subroutine calls.

RPG programs may not be compiled as subroutines. Therefore, RPG programs cannot be called from other languages. You may use another language for any external routines. The best language to use is Cobol, since it supports data types that match those of RPG. If you use other languages, you will have to do extra work to manipulate numeric data. RPG uses the EXIT operator to call the external routine. It passes parameters to it with RLABL or PARM items. Items declared as RLABLs are global, and are not explicitly passed to the routine. All PARM items are passed by reference; i.e. by a pointer to the item. If you are accessing RLABL items in your routine, you must link it with the RPG program file. If you use only PARM items (or none at all), you may link with the program or create an XL file containing your routine(s).

Calling Transact from another language requires the calling program to use an architected call interface (ACI) to allocate the entire Transact data register. Values placed in the data buffer must be correct Transact formats and must be double-byte aligned. If the data buffer of the calling program is not defined exactly as the Transact routine's data register, an error message, such as DATA MEMORY PROTECTION TRAP, is issued and the program terminates.

The data register space allocated to the Transact routine is determined by the data_length parameter of the ACI intrinsic call, not the DATA=Data_Length option of the SYSTEM statement in the called Transact routine. The DATA=Data_Length option of the SYSTEM statement and the ACI data_length parameter differ in how they are declared. The DATA=Data_Length is declared in 16-bit words; the data_length parameter is declared in bytes.

When the called Transact routine uses VPLUS forms, it is the responsibility of the calling program to ensure that the terminal is in character mode prior to the call.

In general, the common data types are properly aligned and so no problems will occur. In Pascal, the system programming extension CRUNCHED form of PACKing will generally produce non-aligned data items within a record; this should be avoided. In Cobol, the SYNCHRONIZED clause may be used to force alignment to standard boundaries, as may the SYNC16 or SYNC32 option. This may be required, as in Cobol, level-01 and level-77 data items outside the Linkage Section are aligned on word boundaries, but the default for other items is byte alignment.

In Transact, use the "%" option to pass a given parameter by byte address; The form "%n" (where the n is either 8, 16, 32, or 64) indicates the bit boundary to align on. When constructing structures, alignment problems may be avoided by placing the largest sized data types first, followed by progressively smaller sized data types. This allows the natural alignment of the individual items and avoids the insertion of padding or filler items.

Calling routines written in other languages from Transact, by default, generates code to make copies of all reference parameters if they are not preceded by "%", "#", or "&". Data is copied to a temporary buffer whose alignment is determined from the type and size of the item passed via the PROC call through which the Transact aligned parameters are passed. However, since this copying is inefficient, the compiler options PROCALIGNED_16/32/64 should be used whenever possible to bypass the copy mechanism.

A future version of Basic will include a new built-in function that returns the byte address of a data item. This will allow the creation of records/structures containing pointers. Until this is available, you can generate a pointer by passing the parameter by reference. If you need to save the pointer for further manipulation by Basic (such as to PACK it into a structure), call a helper function that simply returns the address passed to it, and save this value in an INTEGER variable. You will need to lie to Basic about the type of the return value, and you may need to link the program with PARMCHECK=0. This works because short pointers and integers have the same size and alignment, and are passed the same way in the procedure calling convention.

Fortran COMMON blocks are not shareable with other languages, nor is the Basic COM block linkable to global data in other languages. To access Fortran COMMON or Basic COM data from another language, use a reference parameter to pass the first item of the block to the routine in the

other language. In the called routine, declare the data as a record/structure with data items that correspond to those for the Fortran or Basic common block.

Compilation Options

General:

Frequently, we forget to use the simple compiler options available that provide very useful information for program debugging. TABLE or MAP options can provide symbol lists with type, size, and alignment information. Range checking may be activated for development testing. Automatic initialization may be selected to avoid those nasty undefined value problems in FORTRAN via the $INIT option.

If Shared Globals are used extensively, the default limit on the number of global data references of approximately 2000 may be reached. This limit may be removed with the recently introduced compiler options of $MORE_GLOBALS for Pascal, $CONTROL MOREGLOBALS for Cobol, and the +k pragma for C.

C:

#pragma HP_ALIGN allows you to control the alignment of structure members.

The +u option instructs the compiler to assume that pointers may point to objects that are half-word aligned, and generate code accordingly. This will degrade performance and is not recommended if it can be avoided.

Cobol:

$CONTROL OPTFEATURES=CALLALIGNED flags any parameters not on a 32-bit boundary.

$CONTROL OPTFEATURES=LINKALIGNED improves performance of a Cobol subprogram. Use only if you know that every parameter passed to the subprogram will be 32-bit aligned.

$CONTROL SYNC16 causes the SYNCHRONIZED clause to align on 16-bit boundaries. $CONTROL SYNC32 restores the default, 32-bit alignment.

Transact:

The Transact PROCINTRINSIC option must be used to call an intrinsic when the intrinsic is not declared with DEFINE(INTRINSIC). If this option is not used, the Transact compiler downshifts any procedure names not found in the SYSINTR.PUB.SYS file which may result in the procedure name not being found.

Link Editor Considerations

General:

Dependent libraries

> Dependent libraries will soon be supported on a MPE/iX post-5.0 release. This enhancement will allow a list of library dependencies to be associated with a library. When the library is loaded, its dependent libraries are also loaded. This allows the dependencies to be isolated within the libraries themselves; only the first-level libraries need to be specified when linking or running the program.

Parameter type checking

Different languages have different requirements for parameter type checking. The linker PARMCHECK option can be used to relax the type checking done.

Shared globals

With MPE/iX 5.0, global data symbols can now be visible from an executable library (XL). This allows sharing data between a program file and XLs, but beware that these symbols could now conflict with definitions in the program or in other XLs.

Exception Handling

Pitfalls:

MPE only allows one active trap procedure per process for each type of trap.

In order for some language features to work properly, the trap handler for that language must be active.

Some languages require explicit initialization of the trap handler when a routine is called from a foreign outer block.

The first call to a Basic routine will automatically set Basic trap handling, overriding any previous trap settings.

Recommendations:

If you are using trap handling features in multiple languages, save and restore the traps across each call to a routine in the other language(s).

Details:

While Cobol code is executing, it expects the Cobol trap mechanism to be armed in order for certain exceptions to be processed correctly. In an all-Cobol environment, this is normally done only once, by the main program. The same is true for Fortran and Pascal.

Suppose a Cobol main program calls a Pascal subprogram, but nothing is done to change the trap-handling environment. If a run-time error occurs in the Pascal subprogram, it is likely to be reported incorrectly. For example, a Pascal file I/O error might be reported with the message "No SIZE ERROR phrase (COBERR 747)".

Likewise, a Cobol subprogram called from a non-Cobol outer block is likely to report errors in mysterious ways.

During a call to a Transact routine, arithmetic trapping is always enabled via calls to HPENBLTRAP, XARITRAP and XLIBTRAP, and is reset to the prior state upon return.

To enable Pascal trap handling, call U_INIT_TRAPS prior to calling the first Pascal routine.

To enable Fortran trap handling, call FTN_INITRAP prior to calling the first Fortran routine (if e.g. ON statements are used in the Fortran routines).

To enable Cobol trap handling, call COBOLTRAP to arm the Cobol trap mechanism.

When a non-Basic language calls a Basic routine, the traps for the calling language will be changed to Basic trap handling mode. Basic sets up its trap handling with calls to XCONTRAP, XARITRAP, and XLIBTRAP. This is done during program initialization, after the SYSTEM or SYSTEMRUN statements, or when the BRK function is called with a positive argument. Program initialization occurs when a Basic main program is run, or on the first call to a Basic routine from a foreign outer block.

The Basic BRK function can be used to reset Basic traps if trap handling has been changed by a foreign routine or by explicit calls to the trap intrinsics.

HALT is simulated in Basic by checking for control-y being pressed twice within a short time period. In compiled code, the check for the HALT condition is done at the end of each statement and within certain Basic library routines. Therefore, if the HALT occurs when executing a foreign routine, detection of the condition may be delayed until execution resumes in Basic.

Runtime Factors

Pitfalls:

HFS file names may exceed the previous maximum name length for MPE files.

Certain file names are used by some languages for special purposes.

Special care must be taken when using VPLUS forms with some languages.

Some file types and special I/O handling may not be supported by all languages.

File I/O cannot readily be shared between C and other languages because I/O in C is buffered internally.

Basic may output a terminal status check during initialization.

The Basic Report Writer has no knowledge of output generated by intrinsics or by foreign routines.

Recommendations:

Use MPE file names for maximum compatibility. Most file access routines can handle names up to 86 characters, but sometimes a maximum of 35 characters is used.

Avoid using file names that conflict with those reserved by any of the languages.

Use VPLUS intrinsic calls rather than language form-handling statements if the form must be active across inter-language procedure calls.

Make sure that the terminal is in character mode before calling a Transact routine.

For access to non-standard files, such as KSAM, message IPC, CIO, RIO, etc. use system intrinsics as special open options, control calls, and buffer management are not available in all languages.

Make sure that the I/O buffer is flushed before calling a foreign routine or intrinsic that accesses shared files.

If possible, avoid dependencies on BASIC-compatible terminals in Basic routines that will be called from foreign outer blocks, and set the terminal type in the configuration file to non-Basic.

While a Basic report is active, use only Basic I/O statements to write to the output file.

Details:

I/O in standard C is handled by explicit calls to library functions, which are implemented on top of the MPE/iX file system. Typically, C file I/O is buffered, with the C library maintaining its own buffers, distinct from those used by MPE. When possible, the C library uses mapped file access for reading and writing to disk files. The file buffers and other data structures maintained by C are not common with any other language.

If a Pascal routine is called from a foreign outer block, the input and/or output standard files must be opened in the Pascal routine with the reset and rewrite functions.

If a Pascal file number is passed to a foreign routine, the eof function can be used, upon return, to re-establish the file position, reflecting any changes made to the file while not under control of Pascal.

The following file names should not be used in a Transact program or in a file equation while Transact is running. They could be overwritten without notice:

IPxxxxxx - Transact p-code file

ITxxxxxx - TRANDEBUG file (version A.04.00 and earlier)

IUxxxxxx - TRANDEBUG file (version A.04.02 and later)

OUTPUT - Internal file used by Transact

where xxxxxx is the system name of the Transact program.

When calling a Transact routine, all databases and files specified in the system statement will be opened, whether or not they are accessed by the calling program. No database or file information is preserved during a call to a Transact routine. The calling program has sole responsibility for managing these issues. Transact routines assume the terminal is in character mode on entry, and will unconditionally return the terminal to character mode on exit. Transact maintains its own copy of the VPLUS comarea, which is not accessible outside the Transact runtime system.

In Basic, each compiled routine checks to see if the library has been initialized. If it hasn't, the initialization routine is called. This initialization routine queries the HPBBCNFG.PUB.SYS file, if it exists, to determine defaults for such things as the PRINTER specification and whether the terminal is BASIC-compatible. Be aware that if the configuration file specifies a BASIC-compatible terminal, the Basic

initialization routine will output a terminal status check to query the type of the terminal. This status check can interfere with such things as typeahead and PAD terminals. VPLUS intrinsics and Basic statements should not be mixed to access the same form. Basic maintains its own copy of the VPLUS comarea, which is not accessible outside the Basic runtime system.

The FNUM function can be used to get the MPE file number of a Basic file designator. Be aware that Basic may buffer output until a record is filled.

The POSITION statement will flush the buffer to the file in addition to positioning to a specific record. Basic maintains its own record counter for sequential I/O, so if the file position changes via intrinsic calls or I/O in other languages, a POSITION or direct I/O statement should be used to re-establish the file position. Basic data files (file code 1248) contain extra information about the type of each data item.

Debugging Tips

Pitfalls:

Optimization can cause programs to fail. Either because the program violates one of the assumptions of the compiler, or because the compiler failed to allow for some corner-case condition. Turn off optimization at the start of debugging to avoid chasing these very subtle and hard to isolate problems.

Recommendations:

Utilize the compiler options for symbol lists, range or overflow checking, and initialization.

Turn on range and overflow checking in languages that support it.

For languages that support level 2 optimization, the optimizer can sometimes detect uninitialized variables and give a warning.

C, Cobol, Fortran and Pascal support the XDB and Toolset symbolic debuggers.

Use compiler options to print code offsets and symbol maps for assembly-level debugging. Symbol maps can also help detect inconsistencies with structure layouts that are shared between multiple languages.

Debugging block-mode applications can be done by redirecting I/O to a hardwired terminal or nailed network device.

Details:

Some languages have an option to turn on automatic initialization. If the program works with initialization on and fails with it off, this could indicate the problem is due to an uninitialized variable.

In the linker, an indirect file can contain options to be passed to the link command. Some useful debugging options are:

-v Display verbose messages during linking. For each library module that is loaded, the linker indicates which symbol caused that module to be loaded.

-y <symbol> Indicate each file in which <symbol> appears. Many such options can be given to trace many symbols, but -y must precede each one.

Even when the outer block language does not support the Symbolic Debugger (XDB), routines written in C, Cobol, Fortran and Pascal can be compiled for XDB debugging and linked into the program. This will allow those routines to be symbolically debugged, while the rest of the program can be debugged at the assembly level.

XDB does not currently support debugging code in XLs. Statically link all code you wish to symbolically debug into the program file.

By default, XDB uses the language syntax of the current procedure. This can cause confusion when debugging programs written in multiple languages. For example, in XDB C language mode, the command

p x = 0

will assign 0 to x and print 0, the new value of x. In Pascal mode, it will print 0 if x is not 0, but the value of x will not change.

p $lang

will print the name of the language currently being used. If you explicitly set $lang to a particular language, all debugging commands will use this language syntax regardless of the language of the procedure you are debugging.

RPG has a DEBUG operator which may be used (in the RPG program only, not in the external subroutine) to display indicator settings and the contents of a field wherever it is used in C-specs. This may not be adequate for debugging, particularly in the external procedure. The only other approach is to use NMDEBUG. RPG does not support XDB or Toolset. RPG RLABL global items can be referenced by name in NMDEBUG.

One place where you can get in trouble is in debugging applications that use RSI or VPLUS. Since

these run in block mode, you must redirect I/O to a hardwired terminal before running the program. To redirect I/O for VPLUS applications, get the file name used for the terminal (eg., the RPG program F-spec defining the workstation). File equate this name to the LDEV number of the hardwired terminal. For RSI applications, the name RSIIO must be file equated to the hardwired terminal LDEV.

Transact supports its own debugging subsystem, TRANDEBUG. You must compile your program with the "TRANDEBUG" compiler option to allow debugging via TRANDEBUG. TRANDEBUG can be disabled without recompiling the program by setting the system variable "TRANDEBUG" off via a SETVAR TRANDEBUG, "OFF" command. TRANDEBUG single-stepping returns control after each Transact statement, except in the case of LIST or DATA statements,

which return control after each data item is loaded into the list or data register. TRANDEBUG does not support other programming languages, such as Cobol, Fortran or Pascal, but allows you to invoke NMDEBUG for this purpose.

Basic does not support XDB or Toolset. Debug the Basic portion of your program from the interpreter, or use NMDEBUG or XDB to debug compiled code in assembly mode. To do assembly level debugging, use the COPTION LIST,ID,LABEL options when compiling your Basic program.

LIST - enables ID and LABEL. This is the default.

ID - Prints a table of identifiers, types, and addresses.

LABEL - Prints a table of program line numbers and their code offsets. Assembly code can be dumped during compilation using ;info="#-da" in the compiler runstring.

Pitfalls:

Do not use MARK and RELEASE to free heap space if routines in foreign languages are called. Pascal, C, Fortran, and Basic use the heap to store data that is expected to be preserved between invocations.

Posix Environment

Pitfalls:

New HFS-filenames can cause many problems.

Only the C compiler is supported under the MPE/iX Posix Shell.

Recommendations:

Except for the C language, language development should be done in the MPE/iX Command Interpreter environment.

Details:

POSIX has introduced the HFS-syntax to file names. Valid file names now allow many combinations that were invalid in the past. Especially beware of file names that use language suffixes (like .c or .p) as these file names are often seen as file.group and the groups C or P generally do not exist.

The Pascal compiler does support HFS input and output file names. However, the file names are limited to 132 characters. The Pascal Run-time Library is limited to file names of 255 characters.

Basic, COBOL, Fortran, RPG, and Transact do not currently support POSIX file naming conventions.

BIBLIOGRAPHY

General Information:

Data Types Conversion Programmer's P/N 32650-90015 Oct 1989

Guide

Communicator 3000 MPE/iX P/N 30216-90124 Jan 1995

Compiler Library/XL Reference Manual P/N 32650-90029 Oct1988 Chapter 3: describes procedures for converting numeric data of various types,including floating-point.

Chapter 4: describes procedures for manipulating packed- decimal data in languages that do not support that data type.

General Release 5.0

Chapter 6. Application Development

Chapter 10. Technical Articles

HP Link Editor/iX Reference Manual P/N 32650-90030

HP Precision Architecture P/N 09740-90014

and Instruction Set Reference Manual

New Features of MPE/iX: P/N 32650-90351 Apr 1994

PA-RISC Procedure Calling Conventions P/N 09740-90015 Reference Manual,

Symbolic Debugger/iX User's Guide P/N 31508-90003

Technical Addendum for HP Link Editor/iX, P/N 32650-90476

Trap Handling Programmer's Guide P/N 32650-90026 Dec 1988

Using the Hierarchical File System

Business BASIC/iX:

HP Business BASIC/iX Reference P/N 32715-60001 Oct 1989 Manual

Programming in HP Business BASIC/iX, a documentation file provided with the HP Business BASIC/iX product on the subsys tape, currently named Z00Z715A.HP32715.CLL

C/iX:

HP C/iX Reference Manual (part number 31506-90005), June 1992. Chapter 9: describes data representation and alignment.

HP C Programmer's Guide (part number 92434-90002), August 1992. Chapter 2: discusses data type alignment on various architectures.

Chapter 3: describes how to call other languages and discusses a number of issues with calls to Pascal and FORTRAN.

Cobol II/iX:

HP COBOL II/XL Reference Manual P/N 31500-90001 Jul 1991 Chapter 7: describes data representation. See especially the description of the SYNCHRONIZED clause

and the USAGE clause.

Chapter 11: documents the mechanics of procedure calls. Appendix B: describes the various compiler options.

Appendix H: contains much useful information for the Cobol programmer making calls to and from other languages. See especially the discussion of the Cobol trap mechanism.

HP COBOL II/XL Programmer's Guide P/N 31500-90002 Jul 1991 Chapter 4: contains vital information about calling

sub-programs, including a discussion on calling non-Cobol subprograms, with examples in C, Fortran, and PASCAL.

Fortran 77/iX

HP Fortran 77/iX Reference Manual P/N 31501-90010 Jun 1992 Chapter 8: Interfacing with Non-Fortran Subprograms

HP Fortran 77/iX Programmer's Guide P/N 31501-90011 Jun 1992 Chapter 8: Interfacing with Other Languages

PASCAL/iX:

HP Pascal/iX Reference Manual P/N 31502-90001 Jun 1992

HP Pascal/iX Programmer's Guide P/N 31502-90002 Jun 1992 Chapter 9: External Routines

Chapter 10: Intrinsics

RPG/iX:

HP RPG/iX Reference Manual P/N 30318-90003 Nov 1993 Chapter 8: Calculation Specifications. Examples for PARM, RLABL operators.

HP RPG/XL Programmers Guide P/N 30318-90001 Dec 1988 Chapter 5: Processing Data in an RPG Program.

Transact:

HP TRANSACT Reference Manual P/N 32247-60003 Apr 1994

» Return to original page