

Unix Application Migration

The overall experience of UNIX programmers porting applications from UNIX to OS/390 is that it is as easy as porting between any other UNIX environments. They are happy to consider OS/390 as a UNIX system. Much of the time spent porting to OS/390 is spent on tasks that are common to porting generally.

1. Generic Porting Issues

1.1 Header Files

A header file is the name of a library of functions supplied with the compiler. When a programmer invokes certain functions from his C code, he includes the required header files in the code. Thus, portability of code relies upon consistency of header files across different C products.

Unfortunately, this is not always the case and any port between any two UNIX platforms will inevitably involve a certain amount of time locating functions with slightly different names, or that are declared in different header files. The widespread adoption of UNIX95 standards by application programmers will facilitate this process in the future.

1.2 Make

The make facility is a rule-driven utility for managing the compilation process. The programmer specifies the relationships between programs and make ensures that compiles and links are done correctly and at the right time.

Make implementations differ across UNIX platforms. This means that makefiles themselves need to be ported. That is, code that does not work on OS/390 needs to be identified and changed. The OS/390 make tends to enforce stricter conformance to standards than many other flavors of UNIX, although the differences are smaller with each new release of OS/390.

1.3 C Compiler

Similarly, the OS/390 C/C++ compiler tends to require stricter conformance to language standards than some other compilers. For example, differences occur in handling of pointers, prototyping, castings and datatypes.

During the early stages of a port it can be useful to use compiler options such as `langlvl=extended`, `.POSIX: special target`, and `export_c89_ccmode=1`. All of these will reduce the number of compiler error messages.

As the port progresses, these options should be removed if it is desirable to produce the most portable standards-compliant code. A similar technique, useful in advance of a port, is to change the make settings on the source platform to the strictest available.

Some header file, make and compiler differences identified during porting projects are collated in the redbook *Porting Applications to the OpenEdition OS/390 Platform*, GG24-4473

2. Porting Issues Specific to OS/390

2.1 spawn versus fork Considerations

If your application creates a lot of processes and you want better performance, `spawn()` is the way to go. Similar to `fork()` and `exec()`, `spawn()` runs much faster and saves resources because it does not have to copy the address space. In fact `spawn()` can optionally place the new process in the same

OS/390 address space, even further saving system resources. If your application is multithreaded you must use `spawn()` instead of `fork()`.

If your application is designed to create multiple processes with each running the same program, `spawn()` might not be useful. Many applications rely on having program initialization performed once by the parent process and propagated via `fork()` to all the children processes. The `spawn()` function only propagates a few things like open file descriptors; `spawn()`'s assumption is that the new process will run a different program, not another copy of the same one.

2.2 Portable Header Files

If an application on a UNIX system is not POSIX- or XPG4-compliant, then you may not be able to just move it to an OS/390 UNIX System Services system and expect it to compile. Such applications may include headers that are not supported by OS/390 UNIX System Services application services. Porting an application that does not conform to those standards requires that you inspect all headers that may not exist on an OS/390 UNIX System Services system and determine whether or not the application really requires them. As you know, headers can contain all kinds of things, from macros that simply exist for convenience to prototypes of functions that may or may not exist on a particular UNIX system.

2.3 `sys_errlist`

Note that if you compile and link a program using extern char `sys_errlist`, you will get an informational message:

```
WARNING EDC4011: Unresolved writable static references are detected. FSUM3065
The PRELINK step ended with return code 4.
```

However, the link does not halt and you will end up with a program that is executable (`c89` or `cc` returns 0) but will abend when it tries to access `sys_errlist`. You have to generate a verbose list to find out that it is `sys_errlist` that cannot be resolved.

2.4 Dynamic Link Library

Prior to OS/390 Release 3, you need to use the `-Wl,dll` link-edit option when building a non-DLL application, and just disregard the `.x` file that is produced. The reason: since you are linking in `.o` files compiled for creating a DLL, the linkage editor thinks that your application is itself a DLL.

As of OS/390 R3 (make sure you have APAR OW23744), you no longer need to specify `-Wl,dll`. If you do not specify it, however, you will see a warning message from the linkage editor (which helps to detect that you may have unintentionally exported symbols). By default, `c89` keeps going with warning messages, so your executable will still be built.

In order to use DLLs, programs written in C must be compiled with the `-Wc,dll` option. This option causes different code to be generated, in order to dynamically link to (import) symbols from DLLs. Programs written in C++ are always DLL-enabled, so no special option is needed, and only one style of code is ever generated. Currently, exporting a symbol simply sets a flag in the symbol dictionary entry to indicate that the symbol is exported. So, in C you can create a DLL (which exports symbols) that is not itself DLL-enabled (it does not dynamically link to any DLLs, and so it is itself not compiled with `-Wc,dll`).

The point here is that what is needed to import and what is needed to export are two different issues. It is the exporting of symbols that causes the link-edit to create a `.x` file. This is independent of whether any `.o` files being linked were created from `.c` files compiled with `-Wc,dll`.

So, you need not recompile the `.c` files from your own DLL without `-Wc,dll` in order to statically link. If all the `.c` files are compiled with `-Wc,dll`, but not link-edited with any `.x` files, then the application is DLL-enabled, but just is not using any DLLs.

You also do not need to recompile without `-Wc,exportall` (or `#pragma export`). Prior to OS/390 Release 3, you have to use the `-WI,dll` option, but that is the only change you need to make.

2.5 ASCII-to-EBCDIC Conversion

The main complaint among UNIX programmers is that OS/390 UNIX System Services uses EBCDIC.

2.5.1 Typical Problem Areas

When porting a program to OS/390, an experienced OS/390 UNIX System Services tester says experience has taught him to keep an eye out for these areas where the ASCII to EBCDIC conversion may cause problems:

- Hard-coded ASCII characters in C code as well as shell scripts

Avoid using hard-coded values or depending on the values of characters at all costs. For example, a program might use `'\012'` (octal) instead of `'\n'`. A program might use characters as indices into arrays that were populated using the ASCII values for indices (for example, in C, if `Tab` is an array, `Tab['a']` is equivalent to `Tab[0x61]` in ASCII, but not in EBCDIC), and so on.

- Using the high-order bit of a character for some special purpose

You can do this in ASCII because only 7 bits are necessary for all the printable characters, but that is not true in EBCDIC.

- Assuming the alphabet (a...z) is contiguous

This is true in ASCII, but not in EBCDIC where there are three noncontiguous groups of letters. Even seemingly harmless code like the following probably needs to be changed:

```
char c; for (c='a'; c<='z'; c++) { ... }
```

- Using code generated by `lex` or `yacc`

Often, packages contain C code that was generated by the `lex` or `yacc` utilities. This code will probably contain ASCII dependencies and will not work on OS/390. The code needs to be generated on OS/390 by rerunning the utilities. Note that this may introduce EBCDIC dependencies, making the code less portable to other systems, but at least it will work on OS/390.

For example, `y.tab.c` is typically generated by `yacc` and there should be commands in the package's makefile instructing make how to invoke `yacc` to rebuild `y.tab.c`. There should also be a comment in `y.tab.c` that specifies the source file that `yacc` processed to generate `y.tab.c`.

- Applications that talk to arbitrary remote systems via sockets (such as an FTP client)

These applications typically have to assume that all text they receive is ASCII and they send out all text as ASCII. They have to convert the data locally on the fly.

Consider an FTP client: a user will type a command such as

```
dir foobar
```

The FTP server does not want to see these characters in EBCDIC, so the client must convert the data to ASCII before they are written to the socket. Likewise, if you simply write the data coming from the server to the user's screen, it will be meaningless because it will be in ASCII. The client must first convert the data to EBCDIC. This is true even if the server is running on an EBCDIC system, such as OS/390 or VM.

However, you must be careful to convert only text data. Some applications may mix binary and text in a data stream. For instance, the server might send 2 bytes of binary data preceding a block of text to represent the number of bytes in the block, a cksum value, and so on.

- Code that relies on byte order of data may not be portable. PC systems are "little endian" (that is, the leftmost byte is the most significant), however OS/390 and most UNIX systems are "big endian." This typically affects integer and floating point data. If an application is responsible for transferring such data between platforms, you need to either (1) write data exchange logic or (2) translate to text, transfer as text, and then recreate as binary.

2.5.2 Functions That Support ASCII Input/Output

Standard I/O and String Library Functions: The OS/390 C/C++ run-time library functions support EBCDIC characters. The new libascii package and the C/C++ compiler version V1R3.0 supply a predefined macro, `__STRING_CODE_SET="ISO8859-1"`. This macro provides an ASCII-like application environment on OS/390. You can download our libascii package, which provides an ASCII interface layer for some of the most commonly used C/C++ run-time library functions. libascii supports ASCII input and output characters by performing the necessary `iconv()` translations before and after invoking the C/C++ run-time library functions. The `__STRING_CODE_SET="ISO8859-1"` predefined macro generates ASCII characters, constants, and strings.

Converting text files in an archive: You can set an environment variable in your .profile to handle conversion from ASCII to EBCDIC for text files contained in archives. Here is an example showing how to set an environment variable called A2E and then use it:

```
$ export A2E='-o from=ISO8859-1,to=IBM-1047' $ pax $A2E -rzf foobar.tar.Z
```

"The -o option is not pretty to look at, but once you hide it in a variable, it is easy to use and works perfectly. I have converted millions of bytes of text data this way and have not had a single conversion problem," says an experienced OS/390 UNIX Services tester.

2.5.3 Commands and Functions That Handle Conversion

There are shell commands, TSO/E commands, and C functions that handle ASCII to EBCDIC conversion.

Here are two shell commands that are useful:

- `iconv`

For example, the command: `iconv -f IBM-1047 -t ISO8859-1 words.txt > converted.txt` converts the file `words.txt` from the IBM-1047 code set to the ISO 8859-1 code set and stores it in the file `converted.txt`.

- `pax`

For example, the command: `pax -wf testpgm.pax -o to=IBM-1047,from=ISO8859-1 /tmp/posix/testpgm` backs up the `/tmp/posix/testpgm` directory, which is in the character set CP1047, into an archive file that is targeted to an ASCII character set (ISO8859-1).

The TSO/E commands `OPUT`, `OGET`, and `OCOPY` let you convert files between ASCII and EBCDIC.

The C functions `__atoe()`, `__atoe_l()`, `__etoa()`, and `__etoa_l()` also perform ASCII-EBCDIC conversion.

2.6 libascii: Supporting ASCII Input/Output

2.6.1 Overview

The libascii package helps you port ASCII-based C applications to the EBCDIC-based OS/390 UNIX System Services environment.

The C/C++ run-time library functions support EBCDIC characters. The libascii package provides an ASCII interface layer for some of the most commonly used C/C++ run-time library functions. libascii supports ASCII input and output characters by performing the necessary iconv() translations before and after invoking the C/C++ run-time library functions. Note that not all C functions are supported (see "Limitations" in topic 2.6.3 for additional information).

The OS/390 V1R3.0 C/C++ compiler predefined macro `_STRING_CODE_SET__="ISO8859-1"` generates ASCII characters rather than the default EBCDIC characters. Using this with the libascii code provides an ASCII-like environment.

The libascii package is as thread-safe as the run-time library except where stated under "Limitations" in topic 2.6.3 below.

2.6.2 Floating Point Conversion

The libascii archive also contains four functions to convert between IEEE floating point and S/390 native hex floating point. The OS/390 V1R3.0 C/C++ compiler does not support IEEE floating point. Math operators such as + (add) and / (divide) and run-time library functions such as printf() and sin() that use IEEE floating point numbers will produce undefined results. The main difference between the two floating point formats is that IEEE supports a larger range of numbers, up to 10 to the 308 power. S/390 supports better precision but a smaller range, up to 10 to the 75 power.

2.6.3 Limitations

The libascii interface package is code that we found useful in our IBM porting work, and it is offered "as is" for your use. It is intended to assist in getting your applications running as quickly as possible.

These are some of the known restrictions:

- Not all the C functions are supported. The file libascii.lst contains a list of supported ASCII run-time library routines.

- For some of the supported functions there are known restrictions, as follows:

- getopt():

The libascii getopt() function is not thread-safe. The second argument is changed for a short period of time from EBCDIC to ASCII and then back to EBCDIC.

- The printf() family of functions:

- The %\$n specification is not supported in the format string.

- They are limited to 2048 bytes of output. To increase the size of the strings and output supported, change #define MAXSTRING_a in global_a.h.

- The %\$n specification is not supported in the format string.

- The maximum number of arguments supported is 20.

- They are limited to 2048 bytes of input. To increase the size of the strings and output supported, change #define MAXSTRING_a in global_a.h.

- The interface layer is not NLS-enabled; it only supports conversions between the ISO8859-1 and IBM1040-1 character sets.

2.7 Porting with pthreads

Differences with pthreads and mutexes on OS/390 UNIX System Services exist because OS/390 UNIX System Services implemented the POSIX.4a draft 6 standard rather than the final version, POSIX.1c draft 10.

The book Pthreads Programming by Nichols, Buttlar, and Farrell (ISBN 1-56592-115-1) has a chapter on these differences.

The pthreads standard defines thread-safe variants of existing POSIX functions (for example, `strtok_r()` instead of `strtok()`); however, these are not available under Open Edition. IBM's response is that under OS/390 UNIX System Services, the normal versions are thread-safe so you can use them directly. Because the two variants have differing prototypes, this is a problem if you are porting code that contains the thread-safe variants. You have to either change the code or provide your own versions of the `_r` routines, which map onto the "normal" ones.