Crossing the border

Programming between native z/OS and zOS UNIX Systems Services

Skill Level: Intermediate

David Stephens (stephens@au.ibm.com)
Systems Developer
IBM

16 Sep 2008

For almost 15 years, z/OS® has come with UNIX®, in the form of UNIX Systems Services (USS). However, programming between USS and traditional z/OS (accessing USS services from native z/OS or vice versa) still remains a mystery for most programmers. This article fills in these gaps. It explains to native z/OS programmers how to benefit from UNIX on z/OS, and shows UNIX programmers how to interact with traditional z/OS.

Introduction

In 1994, MVS 4.3 (an old version of z/OS) came out with something new: UNIX. Suddenly UNIX programmers with no z/OS knowledge could port applications from other UNIX systems to the mainframe, and create code in their favorite languages like C and Java™, and all in a familiar environment. It also meant that native z/OS programmers can now use UNIX services to do anything from communicate using TCPIP to access UNIX files, and even call UNIX programs. But today most people still have a concept of a "wall" between UNIX Systems Services (USS) and the more traditional z/OS, believing that an application can only run on one or the other. This simply isn't correct; z/OS has the capability for any application to use the services of both.

This article talks about how to write applications that cross this wall -- a traditional z/OS application accessing USS services, or a USS program accessing traditional z/OS services. This article concentrates on two languages: High Level Assembler (HLASM, which is still used in traditional z/OS) and C (the UNIX programmer's favorite). It also mentions some z/OS-specific issues and techniques relating to C. This article refers to the UNIX part of z/OS as USS (UNIX Systems Services), and the "traditional" non-USS part of z/OS as "native" z/OS.

Creating and compiling C

C programs are really just normal z/OS High-Level Language programs, regardless of where they get built or run. They all need to be created, compiled, and bound. Let's first talk about compiling C code.

The z/OS C compiler is actually both a C and C++ compiler in one, and can be called either from native z/OS or USS. The C compiler on z/OS is the C89 compiler, which UNIX programmers know is a bit older than compilers available on other UNIX systems. You can run this compiler:

- in USS using the `C, C++` or `c89` command (they're all the same)
- in native z/OS using the standard ISPF panels (if installed by your Systems Programmer)
- in native z/OS in TSO/E using the `CC` or `CXX` supplied REXX execs
- in native z/OS in batch

To tell the compiler whether your program is C or C++, this is what you need to do:

- USS: name your program file with the correct extension: `program.c` (lower case C) means it's C, `program.C` (upper case C) means it's C++.
- Native z/OS Batch: Use the C compiler `CXX` runtime option for C++ programs. The C compiler SCCNPRC dataset holds two groups of procedures you can call from batch to compile your code: CBC* for C++, EBC* for C.
- Native z/OS TSO/E: Use the `CC` REXX exec for C, the `CXX` REXX exec for C++

Let's look at some of the issues when using C.

Code location

There's no surprise here, but your C code (and header files) can reside either in a USS file or a native z/OS dataset (PDS, PDS/E, or Sequential, and these can be fixed or variable record formats). This is valid regardless of where you're compiling your code from:

- you can compile C code from a native z/OS dataset in USS; you just need to use the special USS "//" format to specify a native z/OS sequential dataset. Here's an example of a USS shell command to compile the program PGM1 in the PDS `MYHLQ.PGMS.C:`
  `c89 -o pgm1.o "//'MYHLQ.PGMS.C(PGM1)'"`
- you can compile C code from a USS file in batch by specifying a PATH on your SYSIN DD statement, which is similar to the example in Listing 1.
  Listing 1. JCL to do compile C code in USS file

```
//COMPILE EXEC CCNC001,
//SYSIN   DD  PATH='/u/mydir/pgms/pgm1.c',
//             PATHDISP=(KEEP,KEEP),PATHOPTS=(ORDONLY)
```

If you choose to store your code in a native z/OS dataset, you'll probably want to allocate it with DCB settings similar to RECFM=VB, LRECL=256 (particularly if porting it from another UNIX). You don't have to limit code to columns 1 to 72 like COBOL and HLASM.

Accessing z/OS facilities from C

C programs can access many z/OS facilities using the standard C functions, regardless of whether they are running in USS or native z/OS. For example the C function `sleep()` is roughly equivalent to the HLASM macro `STIMER.` The C compiler also gives you some z/OS specific functions like `__cabend()` which is the same as the HLASM `ABEND` macro. All C/C++ functions are documented in the *z/OS XL C/C++ Run-Time Library Reference.*

An interesting thing to note is that you can call USS services (like `getpid()` and `fork()`) from a program that is not running under USS. In this case the address space gets "dubbed" automatically, which is a fancy way of saying that it changes to look like a process to the USS kernel.

If you need to access z/OS control blocks, there's no problem with doing this in C, both in native z/OS and USS. The bad news is that IBM® doesn't provide C header files that map these control blocks. But it's not all bad news: the C compiler also comes with a DSECT conversion utility that can convert HLASM DSECTs to C declarations. Listing 2 is a code sample that gets the z/OS Sysplex Name from the z/OS Extended CVT control block.

Listing 2. Sample C code to access z/OS control blocks

```
             /* -----------------------------------------------------------
        Map z/OS Control Block Structures
        ---------------------------------------------------------- */
     /* --- Cut down PSA Structure - just PSACVT to find our CVT - */
     struct psa {
        int psastuff[4];                /* 4 bytes before CVT Ptr   */
```

```
      struct cvt *psacvt;
      /* Ignore the rest of the PSA */
   };

   /* --- Cut down CVT Structure - just CVTECVT to find ECVT --- */
   struct cvt {
      char cvtstuff[140];             /* 140 bytes before ptr    */
      struct ecvt *cvtecvt;           /* Ptr to ECVT             */
      /* Ignore the rest of the CVT */
   };

   /* --- Cut down ECVT Structure - just ECVTSPLX to get Sysplex */
   struct ecvt {
      char ecvtecvt[8];               /* 8 Bytes before Sysplex  */
      char ecvtsplx[8];               /* Sysplex Name            */
      /* Ignore the rest of the ECVT */
   };

   /* -----------------------------------------------------------
      Variables
      ---------------------------------------------------------- */
   struct psa *psa_ptr = 0;           /* PSA starts at address 0 */
   char sysplex[10];

   /* -----------------------------------------------------------
      Put our Sysplex name into the sysplex variable
      ---------------------------------------------------------- */
   strncpy(sysplex,psa_ptr->psacvt->cvtecvt->ecvtsplx,8);
```

Accessing z/OS files from C on USS

Accessing native z/OS files from C isn't much different from accessing USS files. Normal C file functions like `fopen()`,`fgets()`, `ferror()`,`fwrite()` and `fclose()` all work fine for all native z/OS datasets as well as UNIX files. Things to remember when using these functions are:

- When using the `fopen()` function for native z/OS datasets, you must specify the z/OS dataset name using the "//" format (for example: `//'MYHLQ.MYDATASET'`), or the DDName of a previously allocated dataset using the `DD:`*`ddname`* format. But remember that `fopen()` only works with sequential datasets. If you have a PDS/PDSE, you need to also specify the member name (for example: `//'MYHLQ.MYDATASET(MEMBER)').`
- Remember that native z/OS files are *record* based. The *IBM z/OS C/C++ Run Time Library Reference* sometimes refers to record I/O; it's talking about native z/OS datasets here. So for fixed record formats, you'll get the entire record, including trailing blanks.
- The normal C file functions also work for VSAM files. z/OS also includes some VSAM-specific functions like `flocate()` and `fupdate().`

Calling assembler programs from C

C programs on z/OS (both native z/OS and USS) run within Language

Environment® (LE), a set or runtime libraries that come free with z/OS. Assembler modules usually don't, which means that the linkage (how parameters and calling/return addresses are passed) will be different. So when defining your HLASM program in your C code, you will have to include a `#pragma` statement defining the different linkage. Listing 3 shows a C code fragment defining such a HLASM program.

Listing 3. C Pragma statements to call HLASM programs

```
#pragma linkage (PGM1,OS)         /* Non-LE Linkage      */
extern int PGM1(void *, int *);   /* Function definition */
```

However, if your HLASM program is LE compliant, then this #pragma statement isn't needed.

Apart from this, defining and calling HLASM modules is the same as any other module.

Issues and hints

1. If you're writing C code that needs to compile and run on different UNIX systems including USS, then you'll like the `__MVS__` macro. Using this, you can code `#ifdef` statements for conditional compiles. Listing 4 shows how to do this.

2. The default C/C++ compiler options differ between USS and native z/OS. So, check the options you're using carefully.

3. The z/OS C/C++ compiler also provides a facility that gives you a way of writing C code that doesn't run under Language Environment, which is handy for replacing HLASM exits with C code. It's called the Systems Programming Facility (SPC) for z/OS releases prior to z/OS 1.9, and Metal C for z/OS 1.9 and later.
   Note that using this facility only gives you a subset of the runtime library functions. You can find out more in the *IBM z/OS XL C/C++ Programming Guide* (for SPC) or the *z/OS Metal C Programming Guide and Reference* (for C Metal).

4. Unless you specify the C compiler ASCII option, all C code running in USS is in EBCDIC (whereas every other UNIX system on the planet uses ASCII), so any string constants are treated as EBCDIC. This makes sense when you think that all of USS itself (including shell command input, and results of system function calls) runs in EBCDIC. This is of particular interest if writing Java JNI code; you'll need to translate any

information passed from Java to EBCDIC.

5. When creating a DLL with functions or variables to be externalized (including Java JNI code), you must export the function or variable; this isn't done by default. This can be done using the `#pragma export` directive, or the C compiler `exportall` option to export everything.

6. UNIX C programmers will come across a term that they haven't heard of before when compiling on z/OS: **XPLINK.** Extra Performance Linkage (XPLINK) modules are z/OS modules that have a different way of passing parameters and information between callers. It's new to z/OS and is advertised as being much faster than the older native z/OS linkage. But there's a big catch with XPLINK modules: you generally can't call a non-XPLINK program from an XPLINK program, or vice versa. To do this, you'll need an HLASM "glue" module.
   What's more, XPLINK is only supported for C and Java applications (though you can also write an LE XPLINK HLASM program). But if you want to write a 64-bit program you have no option; you must be XPLINK.

   To create an XPLINK program, you must:

   - Specify the `XPLINK` C compiler option.
   - Specify the `GOFF` C compiler option if compiling in native z/OS.
   - Specify the `XPLINK` binder option if binding in USS.
   - Add the `DYNAM=DLL` and `RENT` binder options if binding in native z/OS (all XPLINK programs must be DLLs).
   - Include the Language Environment side files CELHS001 and CELHS003 if binding in native z/OS. See Listing 9 for example JCL to include these two files.

   You can find out more information about XPLINK in the *IBM z/OS C/C++ Programming Guide* or the *XPLink: OS/390 Extra Performance Linkage* Redbook.

Listing 4. C Code Using __MVS__ for Conditional Compiling

```
        #if defined(__MVS__)
/* (z/OS specific code) */
#endif
```

Creating and assembling HLASM

Native HLASM programmers will be familiar with assembling HLASM using the standard ISPF panels and batch; however, you can also assemble these programs in USS using the `as` command. See the *z/OS USS Command Reference* for more information on this command.

Code location

Just like C, your HLASM code can reside in either a native z/OS dataset, or in a USS file. And just like C, you can assemble code in a PDS from USS, a USS file from batch, or some combination of the two.

Accessing USS functions from HLASM

Like C programs mentioned earlier, HLASM program can access USS services from both native z/OS and USS environments without any preparation; just call the service. HLASM programs access these services using the Assembler Callable Services, documented in the *IBM z/OS Unix Systems Services Programming: Assembler Callable Services Reference.* Listing 5 shows some code that gets the current process ID, and puts it in the fullword PROCESSID field. This code gets the address of the `getpid()` service from the z/OS CSRTABLE control block. The offsets of each service in this control block are documented in the *z/OS USS Programming: Assembler Callable Services Reference.*

Listing 5. HLASM code to access USS functions

```
                L      R15,16                  R15 -> Common Vector Table
        L     R15,CVTCSRT-CVT(15)     R15 -> CSRTABLE
        L     R15,24(R15)             R15 -> CSR slot
        L     R15,276(R15)            R15 = Address of getpid svc
        CALL  (15),(PROCESSID),VL
```

If you do not want to go through control blocks to get the address of the USS service, you have two alternatives:

- Perform a z/OS LOAD of the module with the service you want (BPX1GID in the above example)
- Link to the linkage stub in SYS1.CSSLIB. In this case, the code in Listing 5 would change to that shown in Listing 6. In this example, BPX1GID would point to the stub in SYS1.CSSLIB at link-edit time, which would branch to the relevant service.

Listing 6. HLASM code to access USS functions using CALL

```
                CALL  BPX1GID,(PROCESSID),VL
```

You can find documentation on all available USS services for HLASM in the *IBM z/OS UNIX System Services Programming: Assembler Callable Services Reference*. Native z/OS also provides mapping DSECTS for USS areas in SYS1.MACLIB.

Accessing USS datasets from HLASM

Accessing USS datasets from HLASM is ridiculously easy. You have two choices:

1. Call the relevant USS Assembler Callable Service as explained above
2. Use the native z/OS BPAM macros, such as `OPEN`, `BLDL` and `CLOSE`. See the *IBM z/OS DFSMS Using Data Sets* manual for more information.

By default USS files store information in EBCDIC (so no ASCII; EBCDIC translation is needed).

Calling C programs from Assembler

Calling C programs from HLASM (in both USS and native z/OS) is the same as calling any other High Level Language (HLL) program, which basically means that your HLASM program must be a Language Environment program. To do this you need to adhere to Language Environment standards for register and memory usage, and start and end your HLASM program with some z/OS-supplied macros. However, to confuse you, what macros and standards (and where they're documented) depends on addressing mode and whether your program is XPLINK or non-XPLINK:

- if it is 24- or 31-bit and non-XPLINK, start and terminate your program using the `CEEENTRY` and `CEETERM` macros. See the *IBM z/OS Language Environment Programming Guide* for more information.
- if it is 31-bit and XPLINK, start and terminate your program using the `EDCXPRLG` and `EDCXEPLG` macros. See the *IBM z/OS C/C++ Programming Guide*.
- if it is 64-bit (which means it must be XPLINK), start and terminate your program using the `CELQPRLG` and `CELQEPLG` macros. See the *IBM z/OS Language Environment Programming Guide for 64-bit Addressing Mode* manual.

Issues and hints

- Enterprise COBOL and Enterprise PL/I programs can also be compiled and run within USS. See the respective Programming Guides for more information

Binding programs

In z/OS all HLASM, C, PL/I, and COBOL programs, regardless of where they run, need to be bound (or link-edited) by the z/OS Binder. The binder can be run in either

USS or native z/OS:

- From USS: Use the same c89 function to run both the C/C++ compiler and the z/OS binder. You can run them separately or together from the one call. Binder options are specified in the `-W'L,`*options*' flags of c89.
- From z/OS: Use the standard ISPF panels, or submit a batch job, options well familiar to native z/OS programmers.

Load module and object location

It is commonly believed that modules must reside where they are going to run, in a USS directory for USS, and in a load library/PDSE for native z/OS. But this isn't correct. For USS applications, both the USS libpath (specified in your environment variables) and the native z/OS sequence (Job Pack Area, STEPLIB DD, JOBLIB DD, LPA, and Linklist, in that order) will be searched for a called module. Which is searched first depends on whether your program runs with the POSIX runtime option ON or OFF, libpath first if POSIX is ON, and native z/OS libraries first if POSIX is OFF.

You can set this option using the `#pragma runopts(POSIX(ON))` statement in your C program, the JCL EXEC parameter `PARM='POSIX(ON)'`, or from Language Environment runtime options (set using the `_CEE_RUNOPTS` environment variable).

Similarly, native z/OS applications can reside in a USS file. However, the problem with this is that they can only be called from programs running in USS (and with a USS libpath pointing to that directory), or by a program that invokes the USS `loadhfs` service.

Object modules (modules that have been compiled/assembled but not yet bound) can also reside in a PDS, PDSE, or UNIX library. When binding programs, you can reference objects in any of these.

Calling programs - Static versus dynamic

The question of whether to call another program statically (the called program is link-edited/bound into the same load module) or dynamically (the called program is a separate module that is loaded at runtime) is one that most programmers will have seen before.

Static linking

To statically link a module into your program module, you have a few options, including:

- Add an INCLUDE statement to the binder SYSLIN DD to manually include the module (native z/OS only). For example:

```
INCLUDE SYSLIB(INCMOD).
```

- Specify the binder CALL option, and have the module available in the SYSLIB DD of the binder step (native z/OS only).
- Include the module/object in the USS c89 command; for example, to include `incmod.o` in `pgm1`:
  ```
  c89 -o pgm1 incmod.o
  ```
- Create a USS archive file with all your object files, and include it in the c89 command; for example:
  ```
  c89 -o pgm1 incarch.a
  ```
- Create a USS archive file with all your object files, and use the AUTOCALL statement in the binder SYSLIN DD (native z/OS only); for example:
  ```
  AUTOCALL /u/mydir/incmod.a
  ```

Static linking is also your only option if you have a C program calling a non-LE HLASM program, or a non-DLL program directly calling a DLL module.

Dynamic linking

C can only dynamically link a new load module type called a **DLL** (Dynamic Link Library). DLLs are different from the traditional z/OS load module. They

- must reside in a PDSE or USS file.
- are always reentrant.
- can have names longer than 8 characters.
- can be COBOL, PL/I, LE HLASM, or C modules. But you must use the DLL option when compiling COBOL, PL/I, or C, or the GOFF and RENT options when assembling HLASM.
- can be statically called by COBOL and PL/I.
- can be dynamically linked by PL/I programs using FETCH.
- can be dynamically linked by COBOL programs using CALL, but only if the COBOL program is compiled as a DLL.
- are the only way a C program can dynamically call another program.

To create a DLL, you need to:

- Specify the `DYNAM=DLL` Binder option if binding in native z/OS.
- Specify the option `-W 'l,dll'` if calling the binder in USS.
- Make sure you have a `main()` statement in your code, even if your module holds only functions.
- Include the Language Environment side files CELHS001 and CELHS003 if binding in native z/OS (if 31-bit; CELQS003 if in 64-bit). Listing 9 is some sample JCL that shows how to do this.

The *IBM z/OS C/C++ User's Guide* is the best place to start with DLLs. The last point above talks about **side files**. Side files are files that are automatically created by the binder when binding a DLL (though you can also create it yourself manually). From batch, the side file is sent to the `SYSDEFSD` DD; in USS the binder creates a file ending in 'x' (like `pgm1.x`). This side file is really a file that holds binder instructions telling it the module it needs to call a function. For a module with two functions, it may look like Listing 7.

Listing 7. Side file output from z/OS binder for 31-bit applications

```
IMPORT CODE,'MODULE','My_First_Function'
      IMPORT CODE,'MODULE','My_Second_Function'
```

For 64-bit modules, the same side file would look like Listing 8.

Listing 8. Side file output from z/OS binder for 64-bit applications

```
IMPORT CODE64,'MODULE','My_First_Function'
      IMPORT CODE64,'MODULE','My_Second_Function'
```

When binding a program that dynamically calls a DLL, you need to:

- Include the side file instructions in the SYSLIN DD input to the binder if running in native z/OS batch.
- Add the side file to the c89 bind instruction if running in USS; for example: `c89 -o pgm1 dllfile1.x`.

Otherwise, you'll get unresolved external references when binding your program.

HLASM programmers will be familiar with the LOAD macro for loading a module into storage ready to branch to. However, this will only work for native z/OS modules. To load a module from a USS directory you'll need the `loadhfs` USS service.

Hints and tips

- Modules in a USS file can also be APF authorized using the USS

`extattr` command; for example:
`extattr +a mypgm`.

- When using the binder outside of USS, remember to use the `CASE=MIXED` binder option; otherwise, all your program names will be in upper case.
- If moving a DLL between native z/OS and USS, you need to use the z/OS Binder. Listing 9, Listing 10 , and Listing 11 show sample JCL to move a module between USS and a native z/OS PDSE. Note that the `ENTRY CEESTART` statements in Listing 9 and Listing 10 will be `ENTRY CELQSTRT` for 64-bit modules.
  The SCEELIB members CELHS003 and CELHS001 are Language Environment side files necessary when creating a DLL on native z/OS.

  Non-DLL modules can also be moved using the `OGETX` and `OPUTX` TSO/E commands, and the `mv` and `cp` USS commands.
- From z/OS 1.9, side files can be included in USS archive files.
- It is not necessary to run the binder in USS to create a load module in USS. You can create a module in USS from batch specifying the output directory in a PATH DD statement in the SYSLMOD DD. However, it's harder to create a native z/OS load module when calling the binder from USS.
- When building a DLL, you EXPORT functions/variables that are to be used by another program. You do this using the `#pragma export` statement in your C code, or by specifying the `EXPORTALL` C compiler option.

Listing 9. JCL to move a DLL from USS to a PDSE

```
//LINK    EXEC PGM=IEWBLINK,
// PARM='CALL,MAP,LET,LIST,COMPAT=PM4,DYNAM(DLL)'
//SYSPRINT DD  SYSOUT=*
//SYSLMOD  DD  DISP=SHR,DSN=MYHLQ.LOADLIB
//SYSDEFSD   DD DUMMY
//INPUT    DD  PATH='/u/mydir/pgmname',
//             PATHDISP=(KEEP,KEEP),PATHOPTS=(ORDONLY)
//SYSLIB   DD  DSNAME=CEE.SCEEBND2,DISP=SHR
//SYSLIN   DD  DSNAME=CEE.SCEELIB(CELHS003),DISP=SHR
//         DD  DSNAME=CEE.SCEELIB(CELHS001),DISP=SHR
//         DD  *
 INCLUDE INPUT
 ENTRY CEESTART
 NAME PGMNAME(R)
```

Listing 10. JCL to move a non-DLL module from USS to a PDSE

```
//LINK    EXEC PGM=IEWBLINK,
// PARM='CALL,MAP,LET,LIST'
//SYSPRINT DD  SYSOUT=*
```

```
//SYSLMOD  DD  DISP=SHR,DSN=MYHLQ.LOADLIB
//INPUT    DD  PATH='/u/mydir/pgmname',
//             PATHDISP=(KEEP,KEEP),PATHOPTS=(ORDONLY)
//SYSLIN   DD  *
 INCLUDE INPUT
 ENTRY CEESTART
 NAME PGMNAME(R)
```

Listing 11. JCL to move a module from a PDSE to USS

```
//LINK    EXEC PGM=IEWBLINK,
// PARM='CALL,MAP,LET,LIST,COMPAT=CURRENT,DYNAM(DLL)'
//SYSPRINT DD  SYSOUT=*
//SYSDEFSD DD  DUMMY
//INPUT    DD  DISP=SHR,DSN=MYHLQ.LOADLIB
//SYSLIB   DD  DSNAME=CEE.SCEEBND2,DISP=SHR
//SYSLMOD  DD  PATH='/u/mydir/pgmname',
//             PATHDISP=(KEEP,KEEP),PATHOPTS=(ORDWR,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXU,SIRWXG,SIRWXO)
```

The final word

It's surprisingly easy to cross the border of native z/OS and the USS border. In fact, there's no border there at all. z/OS is one operating system with two different interfaces. So apart from normal problems and hiccups, you'll probably find that the biggest problem accessing the various services will be getting used to the difference in vocabulary between USS and native z/OS.

Resources

Learn

- Redbook: SG24-5992-01: C/C++ Applications on z/OS and OS/390 UNIX focuses on how to move applications written in C/C++ from other UNIX operating systems to z/OS UNIX System Services.
- Redbook: SG24-5991-00 XPLink: OS/390 Extra Performance Linkage describes XPLink, the new high performance linkage option for OS/390.
- Paper: Converting your Language Environment C/C++ Applications to XPLink for 64-Bit
- IBM z/OS UNIX Systems Services Porting Guide provides information that minimizes the effort involved in porting an application to the zSeries platform, and it is also useful for programmers who want to write a new UNIX-style application for the zSeries platform.
- IBM z/OS MVS Program Management: User's Guide and Reference helps you learn about and use the end user interfaces provided by the program management component of z/OS
- IBM z/OS DFSMS Using Data Sets helps you use access methods to process virtual storage access method data sets, sequential data sets, partitioned data sets, partitioned data sets extended, z/OS UNIX files, and generation data sets in the DFSMS environment.
- IBM z/OS XL C/C++ Bookshelf provides titles and links to z/OS XL C/C++ manuals.
- IBM Enterprise PL/I Bookshelf provides titles and links to Enterprise PL/I for z/OS V3R7.
- IBM Enterprise COBOL Bookshelf provides titles and links to Enterprise COBOL for z/OS V4R1.
- IBM z/OS UNIX System Services Programming: Assembler Callable Services Reference describes the features and usage requirements for the z/OS UNIX System Services (z/OS UNIX) callable services.
- IBM z/OS UNIX System Services Command Reference provides information on the shell and utilities feature as well as TSO/E (Time Sharing Option Extensions) commands for using z/OS UNIX System Services (z/OS UNIX).
- IBM z/OS Language Environment Programming Guide provides information to help you manage the run-time environment and write applications that use the Language Environment callable services.
- Browse the technology bookstore for books on these and other technical topics.

Get products and technologies

- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - AIX Forum
  - AIX Forum for developers
  - Cluster Systems Management
  - IBM Support Assistant Forum
  - Performance Tools Forum
  - Virtualization Forum
  - More AIX and UNIX Forums

About the author

David Stephens
David Stephens is a software developer on contract to IBM at the IBM Australian Development Laboratories in Perth, Western Australia. He has worked in z/OS development and Level 3 support of Tivoli products such as TBSM, ITCAM for SOA and TDS. David has a degree in Computer Systems Engineering from the University of Tasmania, and worked for 11 years at several different sites as a z/OS Systems Programmer before contracting to IBM in 2001.