

You can access z/OS UNIX services from batch, TSO/E, or ISPF. This issue targets highlights of batch.

- MVS job control language (JCL) to run shell scripts or z/OS UNIX application programs as batch (background) jobs.
  - Executable files in batch. An *executable file* is any file that can be run as a program. An executable file can be a load module (which is a member of a PDS), a program object (which is either a member of a PDSE or a file in the z/OS UNIX file system), or an interpreted file (such as a REXX EXEC).
  - For a file to be treated as an executable file, it must have execute permission allowed for the invoker.
- BPXBATCH, a utility that can do the following:
  - Run executable files in batch.
  - Run shell commands and executable files from the TSO/E READY prompt.

**JCL support for z/OS UNIX** JCL data definition (DD) statements use a *data definition name (ddname)* to specify the data to be used by the program that you are submitting as a batch job. The ddname is used in two places:

1. In your application program, the ddname refers to nonspecific data, rather than a specific data set name or path name.
  2. In the JCL used to submit the application program as a background job. Here it "binds" the nonspecific reference in the program to a specific data set name or path name.
- You can specify a z/OS UNIX file in the JCL for user-written applications or for IBM-supplied services, such as:
    - DFSMS, Program Management Binder, a prelinker, or a linkage editor
    - BPXBATCH
    - The TSO/E OCOPY command

**The PATH keyword** You can use the PATH keyword on a JCL DD statement to specify the path name for a z/OS UNIX file. When you use the PATH keyword, you can also use these keywords:

- **PATHOPTS** to indicate the access for the file (for example, read or read-write) and to set the status for the file (for example, append, create, or truncate). This is analogous to the option arguments on the **C open()** function.
  - **PATHMODE** to indicate the permissions, or file access attributes, to be set when a file is being created. This is analogous to the mode arguments of the **open()** function.
  - **PATHDISP** to indicate how MVS should handle the file when the job step ends normally or abnormally. This performs the same function as the DISP parameter for a data set.
- NOTE:** if PATHOPTS and PATHMODE are absent from the DD statement, an application needs to supply defaults for the options and mode, or issue an error message and fail.

**The DSNTYPE keyword** There are two related subparameters on the DSNTYPE keyword of the DD statement:

- HFS (hierarchical file system) or zFS (z/OS File System)
- PIPE (named pipe)

**Using the ddname in an application** Instead of using data set names or path names in an application, you can use a ddname; then in the JCL, you associate a specific data set or file with that ddname.

**Note:** The parent process's allocations, for both data sets and files, are not propagated by **fork()** and are lost on **exec()**, except for STEPLIB. You have a choice of two methods for accessing data sets and files in an application:

- The ANSI C function **fopen()**
- The OPEN macro

**The fopen() function** The **fopen()** function recognizes and handles the difference between a ddname associated with a data set (DSN keyword) or with a path name (PATH keyword).

**Example:** Issue: `fopen("dd:FRED", "r+")`  
**Note:** The **fopen()** function takes the ddname FRED, determines if FRED refers to a ddname for a file or a data set, and opens it. **Note:** Once a file is opened, **fread()** and **fwrite()** can access the data.

**The OPEN macro** The OPEN macro can open a z/OS UNIX file specified with the PATH keyword or an MVS data set specified with the DSN keyword. The macro supports DD statements that specify the PATH parameter only for data control blocks that specify DSORG=PS (EXCP is not allowed).

DFSMSdfp supports BSAM and QSAM interfaces to these types of files:

- Regular files
- Character special files (null files only)
- FIFO special files
- Symbolic links

**Note:** You cannot open directories or external links.

**Specifying a ddname in the JCL** In the JCL for a job, you use a DD statement to associate a ddname with the name of a specific MVS data set or z/OS UNIX file.

To specify a file, use the PATH keyword.

**Example:** To associate the path name for the file `u/fred/list/wilma` with the ddname FRED, specify:  
`//FRED DD PATH= u/fred/list/wilma`

At another time, you might specify a different file to be associated with the ddname FRED.

To specify a data set, use the DSN keyword.

**Example:** To associate the data set FRED.LIST.WILMA with the ddname FRED, specify:  
`//FRED DD DSN=FRED.LIST.WILMA,DISP=SHR`

**Note:** At another time, you might specify a different data set to be associated with the ddname FRED.

**Using the submit command** The **submit** command submits JCL from the shell. By using this command you do *not* need to open a TSO session to submit JCL. This command accepts the following as input:

- One or more pathnames
- One or more sequential data set or partitioned data set member names
- Standard input.

**Example:** to submit a job that resides in the z/OS UNIX file `buildjcl.jcl`, enter the following:  
`submit buildjcl.jcl`

**The BPXBATCH utility** BPXBATCH is a utility that you can use to run shell commands or executable files through the batch facility.

- You can invoke BPXBATCH from a batch job or from the TSO/E environment (as a command, through a CALL command, or from a CLIST or REXX EXEC).
- BPXBATCH has logic in it to detect when it is running from a batch job. By default, BPXBATCH sets up the **stdin**, **stdout**, and **stderr** standard streams (files) and then calls the exec callable service to run the requested program. The exec service ends the current job step and creates a new job step to run the target program. Therefore, the target program does not run in the same job step as the BPXBATCH program; it runs in the new job step created by the exec service. In order for BPXBATCH to use the exec service to run the target program, all of the following must be true:
  - BPXBATCH is the only program running on the job step task level.
  - The `BPX_BATCH_SPAWN=YES` environment variable is not specified.
  - The `STDOUT` and `STDERR` ddnames are not allocated as MVS data sets.

**Note:** If any of these conditions is not true, then the target program runs either in the same job step as the BPXBATCH program or in a WLM initiator in the OMVS subsystem category. The determination of where to run the target program depends on the environment variable settings specified in the STDENV file and on the attributes of the target program.

**Restriction:** File and data set allocation considerations vary when a BPXBATCH or BPXBATSL request is processed in the same address space via local spawn or forked to another address space. Allocations for any files and data sets other than **stdin**, **stdout**, **stderr**, or **stdenv** and STEPLIB are not available to a program when BPXBATCH uses **fork()** or **exec** (STEPLIB EXCLUDED) to run a program in another address space. **Note:** Data sets that are allocated in JCL, TSO, or an application *may* conflict with data sets used by BPXBATCH.



**Aliases for BPXBATCH** BPXBATSL, BPXBATA2, and BPXBATA8 are provided as aliases for BPXBATCH that use a local spawn to run in the same address space.

**BPXBATSL** BPXBATSL performs a local spawn, but does not require resetting of environment variables.

**BPXBATSL** behaves exactly like BPXBATCH and allows local spawning whether the current environment is set up or not.

**BPXBATA2 and BPXBATA8** BPXBATA2 and BPXBATA8 are provided as APF-authorized alternatives to BPXBATSL.

BPXBATA2 and BPXBATA8 provide the capability for a target APF-authorized z/OS UNIX program to run in the same address space as the originating job, allowing it to share the same resources, such as allocations and the job log.

**Defining standard input, output, and error streams for BPXBATCH**

z/OS XL C/C++ programs require that the standard streams, **stdin**, **stdout**, and **stderr**, be defined as either a file or a terminal.

Many C functions use **stdin**, **stdout**, and **stderr**. For example: **getchar()** obtains a character from **stdin**, **printf()** writes output to **stdout**,  **perror()** writes output to **stderr**.

**Ways to define stdin, stdout, and stderr** You can define **stdin**, **stdout**, and **stderr** in the following ways:

The **TSO/E ALLOCATE command**, using the ddnames **STDIN**, **STDOUT**, and **STDERR**

**Example:** The following command allocates the z/OS UNIX file `u/turbo/myinput` to the **STDIN** ddname:  
`ALLOCATE DDNAME(STDIN) PATH('u/turbo/myinput') PATHOPTS(ORDONLY)`

**Example:** The following command allocates the MVS sequential data set **TURBO.MYOUTPUT** to the **STDOUT** ddname:  
`ALLOCATE DDNAME(STDOUT) DSNNAME('TURBO.MYOUTPUT') VOLUME(volser) DSORG(PS) SPACE(10) TRACKS RECFM(F,B) LRECL(512) NEW KEEP`

**A JCL DD statement**, using the ddnames **STDIN**, **STDOUT**, and **STDERR**

**Example:** The following JCL allocates the z/OS UNIX file `u/turbo/myinput` to the **STDIN** ddname:  
`//STDIN DD PATH= u/turbo/myinput,PATHOPTS=(ORDONLY)`

**Example:** The following JCL allocates member M1 of a new PDSE **TURBO.MYOUTPUT.LIBRARY** to the **STDOUT** ddname and directs **STDERR** output to **SYSDOUT**:  
`//STDOUT DD DSNNAME=TURBO.MYOUTPUT.LIBRARY(M1),DISP=(NEW,KEEP),DSNTYPE=LIBRARY, //SPACE=(TRK,(5,1,1)),UNIT=3390,VOL=SER=volser,RECFM=FB,LRECL=80 //STDERR DD SYSDOUT=*`

`//SPDERR DD SYSDOUT=*`

**Redirection**, using `<`, `>`, and `>>`

**Example:** Even if **stdout** currently defaults to `/dev/null`, entering the following from the TSO/E command prompt redirects the output of the `ps -el` command to be appended to the file `/tmp/ps.out`:  
`BPXBATCH SH ps -el >>/tmp/ps.out`

**Passing environment variables to BPXBATCH** When you are using BPXBATCH to run a program, you typically pass the program a file that sets the environment variables. If you do not pass an environment variable file when running a program with BPXBATCH, or if the **HOME** and **LOGNAME** variables are not set in the environment variable file, those two variables are set from your logon RACF profile. **LOGNAME** is set to the user name, and **HOME** is set to the initial working directory from the RACF profile.

**Note:** When using BPXBATCH with the SH option (SH is the default), environment variables specified in the **STDENV DD** are overridden by those specified in `/etc/profile` and `.profile` (which overrides `/etc/profile`). This is because SH causes BPXBATCH to execute a login shell that runs the `/etc/profile` script and runs the user's `.profile`.

To pass environment variables to BPXBATCH, you define a file containing the **variable definitions and allocate it to the STDENV ddname**.

The file can be one of the following:

- A z/OS UNIX file identified with the ddname **STDENV**
- An MVS data set identified with the ddname **STDENV**

**Ways to define STDENV**

You can define the **STDENV** environment variable file in the following ways:

• The **TSO/E ALLOCATE command**

**Example:** The environment variable definitions reside in the MVS sequential data set **TURBO.ENV.FILE**.

`ALLOCATE DD(STDENV) DSN('TURBO.ENV.FILE') SHR`  
• A JCL DD statement. To identify a z/OS UNIX file, use the **PATH** operand and specify **PATHOPTS=ORDONLY**.

**Example:** The environment variable definitions reside in the z/OS UNIX file `u/turbo/env.file`.

`//STDENV DD PATH= u/turbo/env.file,PATHOPTS=ORDONLY`

• An JCL in-stream data set

**Example:** The environment variable definitions immediately follow the **STDENV DD** statement.  
`//STDENV DD * BPXBATCH uses two environment variables for execution that are specified by STDENV: variable1=aaaaaaaa -BPX_BATCH_UMASK=0755 variable2=bbbbbbbb -BPX_BATCH_SPAWN=YESINO variable5=ffffff *`

**BPX\_BATCH\_UMASK** allows the user the flexibility of modifying the permission bits on newly created files instead of using the default mask (when PGM is specified).

*Trailing blanks are truncated for in-stream data sets, but not for other data sets.*

• **SVC 99** dynamic allocation, if you are running BPXBATCH from a program

**Example: Setting up code page support in a STDENV file** To enable national language support for BPXBATCH, set the **locale**

- Environment variables to your desired locale in the **STDENV** file.

**Example:** to use the English locale, you could put these lines in the file:

`LANG=En_US.IBM-1047 LC_ALL=En_US.IBM-1047`

After you allocate this file to **STDENV**, you can test it by typing:

`OSHELL echo $HOME`

The path name of your home directory should be displayed, instead of just **\$HOME**.

**Passing parameter data to BPXBATCH** Normally, you pass parameters to BPXBATCH using the parameter string—either in a batch job by using the **PARM=** parameter on the JCL EXEC statement.

• The format of the BPXBATCH parameter string is:

`SH|PGM shell_command|shell_script|program_name [arg1...argN]`

• In a batch job, BPXBATCH only allows up to 100 bytes for the parameter string due to JCL limitations. In a TSO command environment, the maximum length of a parameter string is 32,754 bytes. However, BPXBATCH supports the use of a parameter file to pass much longer parameter data—up to 65,536 (64K) bytes.

• To pass parameters to BPXBATCH using a parameter file, you define a file containing the parameter data and allocate it to the ddname **STDPARM**. The parameter file can be either a z/OS UNIX text file or an MVS data set.

```
//SAMPBPX JOB (55,500,999) MVS 'CLASS=A,
// MSGCLASS=RE REGION=0K NOTIFY=SYSUID
// EXECBPX EXEC PGM=BPXBATCH,REGION=8M,
// PARM=SH is /usr/lib/
// /STDIN DD PATH=/stdin-file-pathname',
// PATHOPTS=(ORDONLY)
// /STDOUT DD PATH=/u/gg/bin/mystd.out',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
// /STDERR DD PATH=/u/gg/bin/mystd.err',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
// /STDENV DD *
TZ=EST5EDT
LANG=C
PATH=/bin:/usr/lpp/java/J1.5/bin.
**
```

sample