
NATIONAL WEATHER SERVICE
OFFICE of HYDROLOGIC DEVELOPMENT

Science Infusion Software Engineering Process Group (SISEPG)
Korn and Bash Shell Programming Standards and Guidelines

Version 1.7

Table of Contents

1. Introduction

The Office of Hydrologic Development (OHD) produces software which NWS Weather Forecast Offices (WFOs) and River Forecast Centers (RFCs) use to create hydrologic forecasts for rivers and streams across the country. OHD also develops and maintains software which runs centrally to acquire and distribute critical data to the WFOs and the RFCs. As many other organizations, software has become a critical component supporting the operations of these forecast offices. Because software plays such an important role, it is essential that it be well-written and maintained.

The OHD Science Infusion Software Engineering Process Group (SISEPG) is developing standards and guidelines to ensure that programmers follow good, widely accepted software development practices when coding. It is believed that this will lead to well-written and better structured programs.

Well-written software offers many advantages. It should contain fewer bugs and run more efficiently than poorly written programs. It also makes it easier for a programmer who was not involved in the development of the software to learn how it works.

Software has a life-cycle. A large part of its life-cycle revolves around maintenance. Software may exist for many years, even decades. Long after the original programmer has moved on, the software will require maintenance in the form of bug fixes and enhancements. The time spent doing this and hence the cost is greatly reduced when the code is developed and maintained according to software standards.

The Office of Hydrologic Development (OHD) produces shell scripting software mainly for the following reasons:

1. To make wrappers that can run binary executable programs.
2. To make test procedures for OHD developed software.
3. To run operational software continuously 24/7.
4. To create in-house configuration management systems to develop OHD executables for each Advanced Weather Interactive Processing System (AWIPS) release cycle.
5. To create miscellaneous user commands for particular needs that may enhance system commands.
6. To perform routine maintenance, such as routine purges, clearing log files, etc…

Many of the current scripts have been written with little interaction with peers. Thus significantly different styles have been used, but they do not necessarily reflect a general shell scripting standard or guideline. This document is intended to establish standards and guidelines for shell script coding. These standards and guidelines are intended to add consistency to coding style and to ensure the code is more readable and maintainable for future keepers of the code. It is important to note that standards are not fixed, but will evolve over time. Developers are encouraged to provide comments and feedback to the:

SISEPG (sisepg@gateway2.nws.noaa.gov).

One further and very important aspect of scripting awareness is that increased coding with scripts would perpetuate the "Unix Philosophy"[‡]. Often compiled code becomes very complex and hard to change when addressing the location of data, the use of pathnames, the use of GUI interfaces, etc...; scripting can be much easier to handle these operations and easier to modify. The actual mathematical and science procedures for forecasting should be isolated in the compiled code in easy to use and maintainable compiled "chunks", but easily tied together by scripts.

2. Shell Internal Documentation

All shell scripts shall contain an internal documentation header at the top of the script. Please refer to the standards for internal documentation in the document **General Software Development Standards and Guidelines** for further details. In addition to these general standards, the documentation shall include:

- Usage statement
- List of input, output, and other environment changes:
  - Input options/arguments
  - Input environment variables
  - Input interactive use (standard input)
  - Output variables
  - Output (or changed) environment variables
  - Output to standard out and standard error
  - Output of error status
- List of required commands (other scripts or programs) called by this script.
- Additional comments such as algorithm description, references, special usage warnings, etc…
- List of any known scripts that can execute this script

The header shall be isolated from code and at the top of the script.

3. Shell Coding Standards

3.1 Structured and Modularized

Shell coding is much like compiled language programming in the code can be organized in structured and modularized fashion. The software engineer shall use structured techniques, such as coding functions when writing shell scripts. In many cases these functions may be able to be re-used by other scripts. Code re-use is a very valuable benefit of structured and modularized coding.

3.2 Indentation

Indentation shall be used to distinguish conditional and control blocks of code. A minimum of three spaces shall be used to indent. Indenting 3 or 4 spaces is generally

[‡] For more information about the “Unix Philosophy” go to http://en.wikipedia.org/wiki/Unix_philosophy

considered to be adequate. Once the developer has chosen the number of spaces in which to indent, it is important this indentation amount be consistently used throughout the script. Consistent indentation will make the script more readable and maintainable
Tabs shall not be used to indent, spaces shall be used.

Bad:

```
for file in $DIR_LISTING
do
    if [ -s $file ]
    then
    echo "$file exists and has a size greater than zero"
    else
      echo "$file does not exist OR has a size of zero"
    fi
  done
```

Better:

```
for file in $DIR_LISTING
do
   if [ -s $file ]
   then
      echo "$file exists and has a size greater than zero"
   else
      echo "$file does not exist OR has a size of zero"
   fi
done
```

NOTE: Special "here-documents" and "cat" commands may require a lack of indentation.

3.3 Inline Comments

Inline comments shall be included to describe a function, block of code, and important lines of code. These comments shall be isolated from executable lines by indentation, blank lines, framing, etc…, so that it's easy to see the code itself. Whatever style is used for inline comments, consistency shall be used throughout the code.

Bad:

```
for file in $DIR_LISTING
do
   if [ -s $file ]
   then
# print the filename and noting the file size is > 0
      echo "$file exists and has a size greater than zero"
   else
# print the filename and note the file size = 0
      echo "$file does not exist OR has a size of zero"
   fi
done
```

Better:

```
# Loop through the directory listing and print the filename and print a
# message noting if the file size is greater than zero or equal to
# zero.

for file in $DIR_LISTING
do
   if [ -s $file ]
   then
      echo “$file exists and has a size greater than zero”
   else
      echo “$file does not exist OR has a size of zero”
   fi
done
```

3.4 Coding Style

A coding style should be developed with a pattern that is consistent throughout the code, so others will know what to expect when reviewing the code.

When modifying code by another author, identify the style and ensure the style continues. In the following example note the inconsistent style of the if-then and if-then-else statements.

Bad:

```
for file in $DIR_LISTING
do
   if [ -s $file ];then
      echo “$file exists and has a file size greater than zero”
      if [ “$file” = “process.log” ]
      then
         mv process.log process.log.old
      elif [ “$file” = “backup.log” ]
      then
         mv backup.log backup.log.old
      else
         cp $file $file.bak
      fi   else
      echo “$file does not exist OR has a file size of zero”
   fi
done
```

Better:

```
for file in $DIR_LISTING
do
   if [ -s $file ]
   then
      echo “$file exists and has a file size greater than zero”
      if [ “$file” = “process.log” ]
      then
         mv process.log process.log.old
```

```
      elif [ "$file" = "backup.log" ]
      then
         mv backup.log backup.log.old
      else
         cp $file $file.bak
      fi
   else
      echo "$file does not exist OR has a file size of zero"
   fi
done
```

3.5 Script Naming

When naming a script, mnemonic names shall be used. Short names should be avoided, as short names may be used in future operating system releases. Use the "whereis" or "man" command on the intended script name to make sure the name is not already used by the operating system.

Bad:

```
Run_dec
```

Better:

```
Run_decoders
```

Bad:

```
Cl_logs
```

Better:

```
Clear_logfiles
```

Bad:

```
cr_bak
```

Better:

```
create_backups
```

3.6 Using Paths

Using paths to other scripts, binary files, or data files may be necessary in scripts, especially if the script is initiated by crontab. If pathnames are needed, define them once as a variable near the top of the executable portion of the script and reference this path variable where necessary. For example a path to Java may be needed. If so, one can define as follows:

JAVA_HOME=/usr/java/jdk/bin

Then somewhere within the script, the Java program may be run by executing:

$JAVA_HOME/java theClass

If and when files are moved to other directories, modifying the script to reflect the change becomes an easier task when using variables for paths.

3.7 Variables and Constants

Variables and constants shall be assigned in a variable and constant section near the top of the script. If the script spawns a child process which needs any of these variables, EXPORT should be used to make these variables available to the child process.
EXPORT $JAVA_HOME=/usr/java/jdk/bin

All variables should be initialized even though scripts do not require that they be initialized. Multiple variables shall NOT be assigned on the same line. Doing so will reduce readability and the script may not be portable.

Bad:

```
name=”Suzy”  age=22  occupation=”student”

echo "My name is $name, I am $age years old, and I am a $occupation"
```

Better:

```
name=”Suzy”
age=22
occupation=”student”

echo "My name is $name, I am $age years old, and I am a $occupation"
```

3.8 Test Statements

Double quotes should be used on all string variables in test statements. This is absolutely necessary if a variable is undefined or null.

Not so good:

```
if [ $HOST = “lx1” ]
then
   statement 1
   statement 2

   statement n
fi
```

Better:

```
if [ “$HOST” = “lx1” ]
then
   statement 1
   statement 2
```

```
   statement n
fi
```

3.9 Reusable Code

Code which is repeated in more than one shell script shall be put into its own file/script. This new script can then be called by any scripts which need this functionality. If the new script is called using the "." or "source" command, the environment variables from the calling script will be maintained.

Consolidating repetitious code into a single file/script will reduce overall lines of code and make the code easier to maintain.

4. Shell Coding Guidelines

4.1 File Naming

A highly consistent naming convention for scripts should be used. A project with many scripts should have a common prefix. Adding .bash or .ksh could prove to be a useful extension to a filename.

A few examples of a naming convention are listed below:

1. Scripts created for testing could end with the suffix "_testit". For example, a script built or modified to test the execution a decoder could be named run_decoder_testit.
2. The main commands that run binary executables could have the prefix "run_"
3. A convention commonly used in OHD is naming scripts by how they are initiated. The scripts which are initiated by cron use the prefix "run_" whereas scripts started manually use the prefix "start_".

4.2 Line Length

It is considered good practice to keep the lengths of source code lines at or below 80 characters. Lines longer than this may not be displayed properly on some terminals and could be difficult to print.

Bad:

```
find /users/hads/shef_reports/abrfc/delivered/via_ftp –name '*' –mtime
+0 –print –exec rm{} \;
```

NOTE: The command above should be considered on one line, the word processor causes the long line to wrap and continue on the next line.

Better:

```
FTP_DIR=/users/hads/shef_reports/abrfc/delivered/via_ftp
```

```
find $FTP_DIR –name ‘*’ –mtime +0 –print –exec rm{} \;
```

4.3 Wrapping Lines

When a statement cannot fit on a single line, a backslash (\) should be used that will allow the statement to continue on the next line. The following principles should be applied:

- break after comma
- break after operator
- prefer high level breaks to low level breaks
- break after switch input
- align the new line with the beginning of the expression from the previous line

Bad:

```
find /users/hads/shef_reports/abrfc/delivered/via_ftp –name ‘*’ –mtime
+0 –print –exec rm{} \;
```

NOTE: The command above should be considered on one line, the word processor causes the long line to wrap and continue on the next line.

Better:

```
find /users/hads/shef_reports/abrfc/delivered/via_ftp –name ‘*’ \
     –mtime +0 -print –exec rm{} \;
```

4.4 Spacing

The proper use of spacing within the code can enhance the readability of a script. Good rules of thumb are as follows:

- a keyword followed by parenthesis should be separated by a space
- a blank space should follow each comma in an argument list

4.5 Variable Declaration and Use

Variables shall have mnemonic or meaningful names that convey to a casual observer, the intent of its use. Global and exported variables should be in uppercase. All variables should be defined near the top of the script and shall be initialized prior to its first use. Variable names should basically follow the same rules as C.

A consistent style for input, output, local, and index variables should be used, especially when using mixed case and underscores since keywords use lowercase letters.

If variables are related, a common prefix or suffix should be used to show the relationship. Variables that store directory names could have the prefix or suffix “Dir” or “DIR”. Variables that store filenames could have the prefix of suffix “Fil” or “FIL”. Variables that store full pathnames could have the prefix or suffix “Pth” or “PTH”.

For scripts that call each other, using similar variable names should be considered.

4.6 Script Output

The "echo" or "printf" statement should be used when outputting data from a Korn or Bash shell script. The "print" statement is not available in Bash and many Korn scripts may be executed as a bash script.

4.7 Meaningful Error Messages

Inevitably scripts will fail for one reason or another. It is important that the developer take the time to handle errors and output meaningful error messages. The error messages should indicate what problem occurred, where it occurred, and possibly the time when it occurred.

When running scripts from crontab, the developer should try to capture the output from the script to a log file. Using the redirection character ">" followed by filename will allow the standard output to be captured to a file.

Example:

```
my_script > my_script.out
```

To also capture the output for standard error, use "2>" followed by a filename. Example:

```
my_script > my_script.out 2> my_script.err
```

To capture all output to one file, use "2>&1". Example:

```
my_script > my_script.out 2>&1
```

NOTE: If the output files are not specified in crontab, the output will be captured in a system e-mail and mailed to the user. A further note, there may be cases when the output from a script is NOT needed due the amount of output OR the frequency in which the script runs. When this case is encountered, the coder may choose to send all output to /dev/null. Doing this prevents the unnecessary use of disk space for storing the output.

4.8 Functions

When writing a script, structured programming techniques should be used. Scripts like other source code should be modularized, making use of function calls.

Functions should be short and written to accomplish a task or small set of related tasks.

Functions can be written using any of the following syntaxes:

```
function purge_files {
```

```
    statement
}

purge_files () {
    statement
}

function foo
{
    statement
}
```

Multiple scripts with the same sets of code should be avoided. The code should be modularized by making separate command or function scripts; or by making code sets in separate files that will run when inserted with a "." or "source" command.

4.9 The . and source Command

Using the "." (dot) or "source" command is an efficient way to include sets of code into multiple scripts. For example, a rather large list of variables, common to many scripts, is a good candidate for a separate file to include with the "." or "source" command.

For example a list of PostgreSQL variables common to many scripts:

```
export PGUSER=jsmith
export PGHOST=`hostname -s`
export PGTZ=GMT
export PGLIB=/usr/share/pgsql
export PGDATA=/var/lib/pgsql
export PGDATABASE=hydro
export PGPORT=5432
```

can be placed in a file called postgres_env in the scripts directory of jsmith. The scripts requiring these variables can load these variables by including:

```
. /home/jsmith/scripts/postgres_env
```

The source command can work in a similar fashion in bash only:

```
source /home/jsmith/scripts/postgres_env
```

Appendix A - Header Template

```
#===============================================================================
# SCRIPT NAME:
# ORIGINAL AUTHOR(S):
# CREATION DATE:
# DEVELOPMENT GROUP:
# DESCRIPTION:
#
# USAGE:
# COMMAND LINE ARGUMENTS:
#     Name    Type     Description
#
# INITIATED BY:
#
# INITIATES/CALLS:
#
# REQUIRED FILES/DATABASES:
# GLOBAL VARIABLES:
#    Name       Type       Description
#
# LOCAL VARIABLES:
#    Name       Type       Description
#
# SCRIPT INPUT:
#
# SCRIPT OUTPUT:
#
# EXIT or RETURN CODES:
#
# FUNCTIONS IN THIS FILE:
#
#===============================================================================
```

Appendix B - Example

```
#===============================================================================
# SCRIPT NAME:          purge_files
# ORIGINAL AUTHOR:      John Smith
# CREATION DATE:        3-21-2006
# DEVELOPMENT GROUP:    HSEB
# DESCRIPTION:
#
# This script will purge files older than 24 hours.
#
# USAGE:   purge_files
#
# COMMAND LINE ARGUMENTS:  NONE
#
# INITIATED BY: crontab
#
# INITIATES/CALLS: NONE
#
# REQUIRED FILES/DATABASES:  NONE
#
# GLOBAL VARIABLES:  NONE
#
# LOCAL VARIABLES:
#   Name           Type       Description
#   SYSTEM       string       The hostname of the machine
#
# SCRIPT INPUT: NONE
#
# SCRIPT OUTPUT:  Output from system commands
#
# EXIT OR RETURN CODES: NONE
#
# FUNCTIONS IN THIS FILE:  NONE
#
#===============================================================================

SYSTEM=`hostname –s`

# Purge files by system.

if [ “$SYSTEM” = “lx1” ]
then
    cd ~/data
    find . -name '*' -mtime +0 -print -exec rm {} \;
    cd ~/rawdata
    find . -name '*' -mtime +0 -print -exec rm {} \;
elif [ “$SYSTEM” = “lx2”]
then
    cd ~/data1
    find . -name '*' -mtime +0 -print -exec rm {} \;
else
    echo “Script cannot purge files on $SYSTEM”
fi
```

References

- *Unix in a Nutshell, O'Reilly*
- *Linux in a Nutshell, O'Reilly*