NCEP Central Operations
WCOSS Implementation Standards

March 17, 2016

Version 10.1

Change logs can be found at
http://www.nco.ncep.noaa.gov/idsb/implementation_standards

I. Introduction

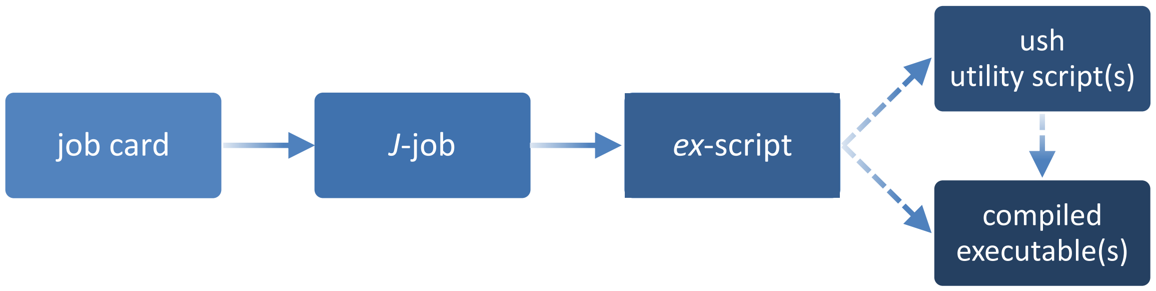
The reliable production and availability of the National Center for Environmental Prediction's (NCEP) guidance products plays a critical role in the mission of the National Weather Service to provide forecasts and warnings "for the protection of life and property and the enhancement of the national economy." This document outlines policies and technical standards that must be met in order to implement operational code or numerical models in the production suite running on the Weather & Climate Operational Supercomputing System (WCOSS) and maintained by NCEP Central Operation's (NCO) Implementation and Data Services Branch (IDSB). WCOSS is currently comprised of three distinct phases, referred to in this document as IBM phase 1, IBM phase 2, and Cray XC40. The coding standards, examples of operational-quality scripts and code, and best practices presented have been established to enable operational stability, efficient troubleshooting and improved Environmental Equivalence (EE) between environments within NCO and between NCO and developing organizations.

II. Workflow

In the production environment, all jobs are scheduled and submitted to the WCOSS resource manager, Platform LSF, by ecFlow. EcFlow is a workflow manager developed and maintained by the European Centre for Medium-Range Weather Forecasts (ECMWF) with an intuitive GUI that is used to handle dependencies, schedule jobs, and monitor the production suite. Each job in ecFlow is associated with an ecFlow script which gets processed to generate a **job card** (a.k.a. *submission script*) whose function is to set *bsub* parameters and much of the execution environment (see Section III-A) and call the *J*-job to execute processing.

The purpose of the ***J*-job** is fourfold: to set up location (application/data directory) variables, to set up temporal (date/cycle) variables, to initialize the data and working directories, and to call the *ex*-script. The ***ex*-script** is the driver for the bulk of the application, including data-staging in the working directory, setting up any model-specific variables, moving data to long-term storage, sending products off WCOSS via DBNet and performing appropriate validation and error checking. It may call one or more **ush** (a.k.a. *utility*) scripts. Additional discussion and examples of the workflow can be found in Appendix A.

All variables relating to the environment in which a job will run must be set, depending on the variable, within the job card or *J*-job. To move a model from development to production, it should generally only be necessary to change the variables exported in the job cards. Downstream scripts should always use the variables established in the *J*-job and should never alter them.



III. Standard Variables, Formats, and Utilities

A. Standard Environment Variables

A standard set of environment variables has been established to simplify the production workflow and improve the troubleshooting process. Table 1 delineates standard environment variables and where they are typically set in the production workflow. They must be used wherever appropriate. In the production environment, the variables with "job card" under "where set" in Table 1 are set in the job card generated by ecFlow. On the Cray XC40 system, several are set by loading the *prod_envir* module. Developers should likewise have a job card for each job which loads any required modules and sets these variables to the correct values prior to calling the *J*-job. Variables that are not used in a given job need not be defined (keep the *J*-job clutter-free!).

Table 1: A list of the standard environment variables

Variable Name	Description	Where Set
RUN_ENVIR	Set to "nco" if running in NCO's production environment. Used to distinguish between organizations.	job card
envir	Set to "test" during the initial testing phase, "para" when running in parallel (on a schedule), and "prod" in production.	job card
NWROOT	Root directory for the application, typically `/nw$envir`	job card
NWROOT*system*	Application root directory on alternate system (*i.e.* `$NWROOTp1`)	job card
job	Unique job name (unique per day and environment)	job card
jobid	Unique job identifier, typically `$job.$$` (where $$ is an ID number)	job card
jlogfile	Log file for start time, end time, and error messages of all jobs	job card
pgmout	File where stdout of binary executables may be written	*J*-job
NET	Model name (first level of com directory structure)	*J*-job
RUN	Name of model run (third level of com directory structure)	*J*-job
PDY	Date in YYYYMMDD format	*J*-job
PDYm#	Dates of a previous day in YYYYMMDD format (`$PDYm1` is yesterday's date, etc.)	*J*-job
PDYp#	Dates of a future day in YYYYMMDD format (`$PDYp1` is tomorrow's date, etc.)	*J*-job
cyc	Cycle time in GMT, formatted HH	job card
cycle	Cycle time in GMT, formatted tHHz	*J*-job
DATAROOT	Directory containing the working directory, often /gpfs/hps/nco/ops/tmpnwprd in production	job card
DATA	Location of the job working directory, typically `$DATAROOT/$jobid`	*J*-job
HOME*model*	Application home directory, typically `$NWROOT/`*`model`*`.v`*`X.Y.Z`*	job card
USH*model*	Location of the model's ush files, typically `$HOME`*`model`*`/ush`	*J*-job
EXEC*model*	Location of the model's exec files, typically `$HOME`*`model`*`/exec`	*J*-job
PARM*model*	Location of the model's parm files, typically `$HOME`*`model`*`/parm`	*J*-job
FIX*model*	Location of the model's fix files, typically `$HOME`*`model`*`/fix`	*J*-job
DCOMROOT	dcom root directory	job card
DCOM	dcom directory for model input data	*J*-job
COMROOT	com root directory for input/output data on current system	job card

COMROOT*system*	com root directory for input/output data on alternate system (*i.e.* `$COMROOTp1` for phase 1 data and `$COMROOTp2` for phase 2)	job card
COMIN	com directory for current model's input data, typically `$COMROOT/$NET/$envir/$RUN.$PDY`	*J*-job
COMOUT	com directory for current model's output data, typically `$COMROOT/$NET/$envir/$RUN.$PDY`	*J*-job
COMIN*model*	com directory for incoming data from model *model*	*J*-job
COMOUT*model*	com directory for outgoing data for model *model*	*J*-job
GESROOT	nwges root directory for input/output guess fields on current system	job card
GESROOT*system*	nwges root directory for input/output guess fields on alternate system (*i.e.* `$GESROOTp1`)	job card
GESIN	nwges directory for input guess fields; typically `$GESROOT/$envir`	*J*-job
GESOUT	nwges directory for output guess fields; typically `$GESROOT/$envir`	*J*-job
PCOMROOT	pcom root directory for outgoing products with WMO headers on current system	job card
PCOM	pcom directory for outgoing products with WMO headers; typically `$PCOMROOT/$NET`	*J*-job
DBNROOT	Root directory for the data-alerting utilities	job card
SENDECF	Boolean† variable used to control ecflow_client child commands	job card
SENDDBN	Boolean† variable used to control sending products off WCOSS	job card
SENDDBN_NTC	Boolean† variable used to control sending products with WMO headers off WCOSS	job card
SENDCOM	Boolean† variable to control data copies to `$COMOUT`	job card
SENDWEB	Boolean† variable used to control sending products to a web server, often ncorzdm	job card
***model*_ver**	Current version of the model; where *model* is the model's directory name (*e.g.* for `$NWROOT`/gfs.v12.0.0, `gfs_ver=v12.0.0`)	version file
***shared_directory*_ver**	Current version of the *shared_directory* (*e.g.* for the gsi shared code in `$NWROOT`/gsi_shared.v5.0.1, `gsi_shared_ver=v5.0.1`)	version file
***module*_ver**	Version of module *module* which used by the current job	version file
KEEPDATA	Boolean† variable used to specify whether or not the working directory should be deleted upon successful job completion.	job card

†boolean variables are set to "YES" or "NO" (all caps)

B. File Name Conventions

Standard file naming conventions should also be used. File names should not contain uppercase characters or the date (the directory in which the file resides will contain the date). File names should indicate the name of the model run, the cycle, the type of data the file contains, the resolution of the data (if applicable), other data related elements, the three-digit forecast hour the data represents (if applicable), and the file type. Please observe the following:

1. Use periods to separate categories and use underscores to separate words within the same category
2. Use a "p" in describing a "point" within a grid resolution. Ex. 0.25 = 0p25
3. Include an "f" in front of the forecast hours

4. Pad forecast hours with zeros so that all files have the same number of digits
5. Output file names should be consistent across environments and application versions, so variables such as `$job`, `$envir`, and `$model_ver` should not be used to define file names.

Filename format for files in **com**:

model.t*HH*z.*var_info*.f###.*domain*.*format*

Example filenames for files in **com** (*HH* is the cycle/hour):

rtofs_glo.t*HH*z.std.f180.west_conus.grib2
aqm.t*HH*z.8hr_o3.227.grib2
sref.t*HH*z.mean_3hrly.pgrb243.grib2

Filename format for files in **pcom**:

format.*model*.t*HH*z.*awp_var_nfo*.f###.*domain*

Example filenames for files in **pcom**:

grib2.aqm.t*HH*z.08hr_o3.227
grib2.akrtma.t*HH*z.2dvaranl.198
grib2.sref.t*HH*z.spread.212

C. Production Utilities

It is imperative that all production code and scripts broadly employ error checking to catch and recover from errors as quickly as possible. The context of the error should be communicated as descriptively as possible and prefaced with “WARNING:” or “FATAL ERROR:”. Failures should not be allowed to propagate downstream of the point where the problem can first be detected. The following utilities should be used to assist in accomplishing these tasks. The below utilities are accessible with the *prod_util* module. This module will prepend the directory containing all production modules to your environment’s PATH variable and define other useful environment variables. See Table 6 (in Appendix B) for variables and their descriptions. The module is loaded in all production ecFlow scripts and should be loaded in development job cards as well. See Appendix A for examples of these utilities in use.

prep_step

`prep_step` unsets the FORT## variables used to pass unit assignments to FORTRAN executables. Since there may be multiple FORTRAN programs running in a job, these variables must be reset before each program execution.

startmsg

`startmsg` posts the start time of a program to `$jlogfile`

postmsg

`postmsg` writes a message to a log file. The first argument is the log file name and the second is the message. In general, `$jlogfile` should be specified as the log file.

err_chk

`err_chk` is used to check and handle the `$err` variable which has been set to a program's return code and exported into the environment. If `$err`=0, the end time of the program is posted to the log file and job execution continues. If `$err` is non-zero, the contents of the file *errfile* and `$pgmout` are written to the output file, the time of the error is logged, and the job is aborted.

err_exit

`err_exit` will write the contents of `$pgmout` to the output file, write an error message with the time of the error, and abort the job. It accepts an error string as input to which it will prepend "FATAL ERROR."

cpreq

`cpreq` has the same usage as the standard `cp` command. It is used to copy files that are essential to the application. If the copy is unsuccessful then a FATAL ERROR will be posted to `$jlogfile` and the output file and the job will abort immediately.

cpfs

`cpfs` has essentially the same usage as the standard `cp` command with the limitation that it may only copy one file at a time (no globbing). It is used to ensure downstream applications will not attempt to copy or read a partial file. It is most useful for copies across file systems or for very large files.

```
        cpfs $COMIN/$file $new_file
```

will execute the following:

```
        cpreq $COMIN/$file $new_file.cptmp
        $FSYNC $new_file.cptmp
        mv $new_file.cptmp $new_file
```

compath.py

The `compath.py` utility is used to discover the current absolute path of a given **com** directory and is mainly used to set COMIN variables in *J*-jobs. As models move from one system on WCOSS to another, this will assist in managing data localization. `compath.py` accepts the relative path of the directory you wish to use data from as an argument; the corresponding absolute path is returned:

```
        COMIN=${COMIN:-$(compath.py $NET/$envir/$RUN.$PDY)}
        COMINm1=${COMINm1:-$(compath.py $NET/$envir/$RUN.$PDYm1)}
        COMINgfs=${COMINgfs:-$(compath.py gfs/$envir/gfs.$PDY)}
        COMINarch=${COMINarch:-$(compath.py arch/prod/syndat)}
```

To use non-production data, in the job card set the `$COMPATH` environment variable to a list of absolute paths. `compath.py` will search those paths for a match before defaulting to production data.

```
        export COMPATH="$COMROOT/nco:/dev/noscrub/First.Last/com/gfs"
```

mail.py

When nonfatal errors occur that may impact the quality of the model output, such as when backup data is used, it is important to notify the appropriate parties so that the error can be addressed. The `mail.py` utility can assist by sending an e-mail notification from any node on the system. To notify production staff of a nonfatal but significant issue with a production job, one might execute:

```
msg="WARNING: Primary data source unavailable. Backup data is being used."
echo "$msg" | mail.py
```

To copy someone, use the "-c" flag:

```
echo "$msg" | mail.py -c first.last@noaa.gov
```

Run "`mail.py -h`" after loading the *prod_util* module to see additional options. Note that e-mail is only sent in jobs run by NCO. Jobs run by others will merely print the message to stdout.

getsystem.pl

`getsystem.pl` simply tells you which WCOSS system you are on. Table 2 shows what you can expect to receive when running this utility on a given system with a given set of option flags:

Table 2: getsystem.pl output

	Tide Phase 1	**Tide Phase 2**	**Luna**	**Gyre Phase 1**	**Gyre Phase 2**	**Surge**
`getsystem.pl`	Tide	Tide	Luna	Gyre	Gyre	Surge
`getsystem.pl -p`	Tide-p1	Tide-p2	Luna-XC40	Gyre-p1	Gyre-p2	Surge-XC40
`getsystem.pl -t`	IBM	IBM	Cray	IBM	IBM	Cray
`getsystem.pl -tp`	IBM-p1	IBM-p2	Cray-XC40	IBM-p1	IBM-p2	Cray-XC40

D. Date Utilities

The following utilities are used to manage dates in the production suite. They must be used wherever current dates are employed to enable proper scheduling and ensure that all jobs work as expected when crossing over to a new year. The following date utilities are accessed by loading the *prod_util* module.

finddate.sh

Given a date, `finddate.sh` will return a date (in YYYYMMDD format) a specified number of days before or after the given date. It may also provide a sequence of dates leading to the specified number of days before or after the given date. Example 1 shows how to use `finddate.sh`. This utility does not work for usage spanning more than two calendar months!

Example 1: Using finddate.sh

Script

```
#!/bin/sh
module load prod_util
PDY=20160101

# Single date example
ten_days_ago=$(finddate.sh $PDY d-10)
ten_days_ahead=$(finddate.sh $PDY d+10)

# Sequence example
last_four_days=$(finddate.sh $PDY s-4)
next_four_days=$(finddate.sh $PDY s+4)

echo "Today's date is $PDY"
echo "The date ten days ago was $ten_days_ago"
echo "The date in ten days will be $ten_days_ahead"
echo "The last four days were $last_four_days"
echo "The next four days are $next_four_days"
```

Output

```
Today's date is 20160101
```

```
The date ten days ago was 20151222
The date in ten days will be 20160111
The last four days were 20151231 20151230 20151229 20151228
The next four days are 20160102 20160103 20160104 20160105
```

ndate

`ndate` is accessible by the variable $NDATE once the *prod_util* module has been loaded. `ndate` is a date utility that will return a date in YYYYMMDDHH format. Given no arguments, it will return the current date/hour. `ndate` takes up to two arguments, namely `fhour` and `idate`:

```
ndate [fhour [idate]]
```

`fhour` is a forecast hour (may be negative) and defaults to zero. `idate` is the initial date in YYYYMMDDHH format and defaults to the current date. Example 2 shows how to use `ndate`.

Example 2: Using ndate

Script

```
#!/bin/sh
module load prod_util

PDYHH=$($NDATE)

# Single date example
ten_days_ago=$($NDATE -240 $PDYHH)
ten_days_ahead=$($NDATE 240 $PDYHH)

# cycle examples
next_cycle=$($NDATE 06 $PDYHH)
prev_cycle=$($NDATE -06 $PDYHH)

echo "Today's date and cycle is $PDYHH"
echo "The date ten days ago was $ten_days_ago"
echo "The date in ten days will be $ten_days_ahead"
echo "Six hours from now will be $next_cycle"
echo "Six hours ago was $prev_cycle"
```

Output

```
Today's date and cycle is 2016010112
The date ten days ago was 2015122212
The date in ten days will be 2016011112
Six hours from now will be 2016010118
Six hours ago was 2016010106
```

setpdy.sh

`setpdy.sh` creates a file PDY that is sourced to export the standard date variables PDYmn_m, PDYm{n_m-1}, PDYm{n_m-2}, …, PDYm2, PDYm1, PDY, PDYp1, PDYp2, …, PDYp{n_p-2}, PDYp{n_p-1}, PDYpn_p. By default, n_m and n_p are 7 but can be altered by providing alternate numbers as input parameters. The variable `cycle` must be set (in 't*HHz*' format) prior to execution. The default date is the current day's date as defined in the file /com/date/$`cycle`, but it can be overridden by setting the variable PDY prior to execution. The date files in /com/date are set at 11:30 UTC and 23:30 UTC. At 23:30, the date files for cycles 00–11 are incremented to the next day. At 11:30, the date files for cycles 12–23 are likewise advanced. Therefore, if you were to set `cycle` to t12z and run `setpdy.sh` between 00:00 and 11:30, you would get a PDY file centered on the previous day's date. Example 3 shows how to use `setpdy.sh`.

Example 3: Using setpdy.sh (assuming current date is 20160101)

Script

```
#!/bin/sh
module load prod_util
export cycle=t12z

setpdy.sh 8 3
. PDY

echo "Yesterday's date was $PDYm1"
```

Contents of file PDY

```
export PDYm8=20151224
export PDYm7=20151225
export PDYm6=20151226
export PDYm5=20151227
export PDYm4=20151228
export PDYm3=20151229
export PDYm2=20151230
export PDYm1=20151231
export PDY=20160101
export PDYp1=20160102
export PDYp2=20160103
export PDYp3=20160104
```

Output

```
Yesterday's date was 20151231
```

E. GRIB Utilities

GRIB is a data format commonly used across the production model suite at NCEP and in Numerical Weather Prediction worldwide. NCO supports several utilities responsible for manipulating GRIB data. These utilities are accessible in production via the *grib_util* module. The module will define numerous environment variables. See Table 6 (in Appendix B) for all variable definitions and descriptions of each utility. The module must be loaded in the job cards of jobs using GRIB utilities:

```
module load grib_util/$grib_util_ver
```

IV. Standards

A. General Application Standards

Diagnosing failures quickly is a necessary component of maintaining a suite of products that boasts a greater than 99% on-time delivery rate. To end, all code should be scrutinized for both stability and ease of troubleshooting. It is not practical to discuss all of the steps that can or should be taken to write operational-quality code, but here are some things that should be considered:

i. Notification of use of backup data

For scripts that have a secondary data source to be used when the primary data is not available, the script should include a message that indicates the primary data is not available and backup data is being used. If continued use of backup data will result in a degraded product, the developer should work with NCO's SPA team to include code to notify the appropriate parties when primary data is unavailable. The `mail.py` utility can be useful in this regard.

ii. Descriptive error messages

Fatal errors should print a descriptive message beginning with "**FATAL ERROR**:". Warnings or non-fatal error messages should be prefaced with "**WARNING**:". As with executable code, error messages in scripts should be written so that if an issue arises, the context of that error or failure is communicated as early and as clearly as possible.

iii. Appropriate modes of failure

An executable should not terminate abnormally with a segmentation or memory fault for errors that are discoverable/trappable. For example, lack of input data should be handled either in the script before the executable runs, or by the executable if checking in the script is not practical.

iv. Minimize the time it takes to re-run a failed job

In places where restarts can be applied to save time when recovering from a failure, they should. Long running jobs that have multiple executable calls might be a good candidate to break into two smaller jobs so that if a failure occurs, only the problem part need be re-run and the time to completion is shorter.

v. No background processing

LSF loses control of processes when they are put in the background. Therefore, background processing must be avoided.

vi. No external-pointing symlinks

Symbolic links to resources outside of the *application directory* or *package* (*i.e.* links to absolute paths) are not allowed within the package. When external resources are required, their paths should be obtained from production module variables (when available) or defined as variables in the *J*-job and used wherever the external resource is needed.

vii. Working directories

Working directories should contain a unique identifier (pid) unless there is an application need to share the directory across multiple jobs (*e.g.* a forecast job writing output that is needed by a post job running in parallel). Working directories should be removed upon successful completion of the run. All data that is needed for longer than one cycle should be copied to $COMOUT, $GESOUT or $PCOM.

viii. Data of opportunity

It is acceptable to use data from a server or other source that is not supported 24/7. However, the application cannot fail when this data is missing. Appropriate notification of use of backup data should be made (see above) and the job should continue with other operationally-supported input data.

Source code and scripts should be annotated with information that may help staff remedy a problem if something goes awry. In some cases, too much information is as bad as none at all. We ask that you use your best judgment to include information that will be of the most help in troubleshooting potential issues. Example 4 shows a suggested format for a documentation block (DOCBLOCK).

Example 4: DOCBLOCK template*

```
# Program Name:
# Author(s)/Contact(s):
# Abstract:
```

```
# History Log:
#   <brief list of changes to this source file>
#
# Usage:
#  Parameters: <Specify typical arguments passed>
#  Input Files:
#    <list file names and briefly describe the data they include>
#  Output Files:
#    <list file names and briefly describe the information they include>
#
# Condition codes:
#    < list exit condition or error codes returned >
#    If appropriate, descriptive troubleshooting instructions or
#    likely causes for failures could be mentioned here with the
#    appropriate error code
#
# User controllable options: <if applicable>
```

* Use appropriate comment indicator (#, !, or //) where appropriate.

B. Compiled Code (C or FORTRAN source)

1. Compiled code must be written in either C/C++ or FORTRAN.
2. C and FORTRAN compilers must be the default Intel version or higher (`icc` and `ifort`) on IBM Phase 1 & 2 and the default Intel or Cray versions or higher (`cc`, `CC`, and `ftn`) on the Cray XC40.
3. All libraries must be approved for production use. On the Cray XC40 system, approved libraries are found by running "`module avail`" and looking under the /usrx/local/prod/modulefiles and /gpfs/hps/nco/ops/nwprod/lib/modulefiles listings. Makefiles should include compilers and libraries using variables defined in modules:

   Within the build script or build module in the parent sorc directory:

   ```
   module load PrgEnv-intel/5.2.56
   module load w3nco-intel/2.0.6
   ```

   Within the makefile:

   ```
   LIBS = ${W3NCO_LIB4}
   ndate: ndate.f
          ftn -o ndate ndate.f $(LIBS)
   ```

   A build modulefile must be provided for more complex builds. See Example 11, Example 12, and Example 13 in Appendix A for an example build script, modulefile, and makefile, respectively.
4. In order for certain errors to be trapped early in the build process, it is recommended that the check_prereqs target be added to all makefiles:

   ```
   check_prereqs:
         /gpfs/hps/nco/ops/nwprod/spa_util/check_libs.bash $(LIBS)
         /gpfs/hps/nco/ops/nwprod/spa_util/check_incs.bash $(INC)
   ```

5. Do not specify absolute paths to executables, libraries, or any other products inside the makefile. With few exceptions, paths should be set by a module.
6. Code should compile without warnings.
7. Errors must be caught as early as possible and the context of the error should be communicated clearly. Failures should not be allowed to propagate past the point where the problem is first detectable.

8. FORTRAN Logical Unit Number (LUN) Assignments:

   In code that uses static units, and where the flow of operation is simple, please make an effort to use a standard or consistent assignment strategy. We understand that in some situations, source code is used by a community of scientists and it can be impractical to assign specific unit numbers to files, but it is useful to have a consistent standard for all input and output wherever possible to provide a means to quickly understand how data is being used.

   - Units **11–49** for all **input** files
   - Units **51–79** for all **output** files
   - Units **80–94** for all temporary **work** files, written and used within in the same program

   Except for work files, the same unit number should never be used for both input and output by the same program. Users should associate filenames to unit numbers in the script prior to program execution. On the WCOSS, users should use the environment variables FORT*k*, where *k* is a two-digit number. Filenames should never be hardcoded in the source.

   Example:

   ```
   export FORT11=inputfile.tbl
   export FORT60=outputfile.grb
   ```

C. Interpreted Code (bash, ksh or perl scripts)

Each “job” is associated with a single *J*-job, located in the **jobs** subdirectory. The *J*-job sets up the environment and calls an *ex*-script script located in the **scripts** subdirectory. All *J*-jobs should follow the naming convention JAAAAA: all capital letters beginning with the letter ‘J’ with no extension. *J*-jobs must use Bash (/bin/bash or /bin/sh, the latter invokes Bash in POSIX mode on WCOSS) or Korn Shell (/bin/ksh). *Ex*-scripts and utility scripts may be written in Bash, Korn shell, Perl, or Python. *Ex*-scripts should follow the naming convention exaaaaa.sh: all lowercase beginning with the letters ‘ex’ and ending with the appropriate extension (‘.sh’, ‘.pl’, ‘.py’). Any sub-scripts to the *ex*-script will be located in the **ush** subdirectory, be named in all lowercase letters *not* beginning with the letters ‘ex,’ and should end with the appropriate extension. Underscores are permitted in all file names.

Please also observe the following points:

1. Enable debug logging at the top of each *J*-job:
   ```
   export PS4=' $SECONDS + '
   set -x
   ```
2. Utilize standard environment variables and utilities (See Section III).
3. Each block of dbnet alerts must be wrapped with logic testing whether the variable $SENDDBN or $SENDDBN_NTC, as applicable, is set to “YES”.
4. Each execution of a C or FORTRAN code must be wrapped with the production utilities `prep_step`, `startmsg` and `err_chk`.
5. Any executions that print verbose output (more than 100 lines or so per execution) should redirect standard out to `$pgmout` and standard error to *errfile*:
   ```
   $EXECmodel/$pgm >> $pgmout 2> errfile
   ```

6. Production utilizes a centralized cleanup of directories in /com and /nwges. Production scripts should not remove directories at the /com/$NET/$envir/$RUN.$PDY level.
7. Any output written to **pcom** should be named in such a way that the files are overwritten with each subsequent run from day to day.
8. Remove all references to developer work areas and all development tools (benchmarking, etc.) before submitting to IDSB.
9. If your application should continue if a preceding step fails, it should be documented in a comment in the script just before (or after) the relevant part is called and a descriptive "WARNING:" message printed to stdout and posted to the `$jlogfile` via `postmsg`.
10. Never write to **dcom**!

Reference Appendix A for commented examples of a version file, ecFlow script, *J*-job, *ex*-script, modulefile and makefile.

V. Dataflow

Distributed Brokered Networking (DBNet) is used to disseminate products operationally from WCOSS. DBNet is a series of server/client daemons that are controlled by table and key relationships. To disseminate a product, jobs running on WCOSS make a call to the dbn_alert executable which makes the DBNet software aware of the new product. Then, based on entries in several different tables, the product can be sent to one or more external servers. The NCO Dataflow Team is responsible for maintaining DBNet and needs to be coordinated with in the event any new alert call is added or if an existing alert is changed. All DBNet alerts must be wrapped in a check for $SENDDBN (or $SENDDBN_NTC) equal to "YES".

```
$DBNROOT/bin/dbn_alert MODEL PMB_GB2 $job $COMOUT/$outputfile
```

Field	Description
Type [MODEL]	Generic data type
Subtype [PMB_GB2]	Specific data type under the generic type
Job Name [$job]	Name of the process that alerted the file, this is only used in the log output. It can be helpful when trying to identify the job that called `dbn_alert`
File [$COMOUT/$outputfile]	File to be alerted; must include the full path.

VI. Code Delivery and Vertical Structure

All components of an application to be run in the NCO production environment must be delivered to IDSB's Senior Production Analysts (SPA) via subversion. When modifying an application that is already in production, always begin with the most recent production version at https://svnwcoss.ncep.noaa.gov/*MODEL*/tags/.

A. Source Code Compilation (C or FORTRAN)

1. The directory structure, compilation scripts, makefiles, and documentation for building should be understandable to someone unfamiliar with the specifics of your model.
2. Do not deliver pre-built executables or libraries to IDSB. It is the SPA's responsibility to build all code before it is run in production.
3. If more than one executable is to be built, divide the source files into sub-directories according to the executable they produce (unless multiple executables share a large portion of their code base in which case sub-directory sharing is allowed). The name of each source directory should be the name of the executable it produces plus the appropriate extension (.cd or .fd for C or FORTRAN code, respectively).
4. All source code must be delivered with a modulefile and/or build script used to set up the build environment. It should define the compiler and its version (by loading the appropriate versioned compiler or 'PrgEnv' module), specific library versions, and all other external files used to compile the application. An example modulefile can be found in Example 12 of Appendix A. Creating symbolic links to external resources (*i.e.* to absolute paths) is not allowed.
5. It is preferable that each source code directory have a makefile that does everything needed to build the executable. For example, global_fcst.fd would contain FORTRAN code and a makefile to produce the global_fcst executable. The basic '`make`' command should not move the compiled binary; however, '`make install`' may do so. Example 13 of Appendix A contains an example.
6. Use a readme file in the source directory to explain the build process, particularly if it requires any interaction or if it is non-standard in any way; for example in situations where a makefile produces more than one executable. Clear, concise instructions (see Example 10 in Appendix A) will reduce confusion and errors if it becomes necessary to rebuild the executable quickly.

B. Directory Structures

All components of an application to be implemented into the production environment are required to be in vertical structure, where, with the exception of system or standard production libraries and input data, all of the files required to completely build and run the jobs are contained in an application-specific package. The package must contain all *J*-jobs and *ex*-scripts specific to the model and should be named with the following format: *model*.v*X.Y.Z* (*e.g.* gfs.v12.0.1). Files should be organized into sub-directories according to their type (see Table 3). If there exists code, scripts or other files shared between multiple models then they should reside in a separate shared package (*e.g.* gsi_shared.v5.0.0). Shared packages should not contain *J*-jobs or a jobs sub-directory.

Table 3: Package Sub-directories

Subdirectory	Description
doc	release notes or other documentation
jobs	*J*-jobs
scripts	*ex*-scripts
ush	utility scripts (ush-scripts)
sorc	source code
exec	binary executables
parm	parameter files

fix	fixed fields, tables or other static input data
lib	model-specific libraries
ecf	ecFlow scripts and definition files (developers not responsible for this directory)
gempak	all gempak-related files

Table 4 lists the primary data and application directories used within the WCOSS NCO production environment. On the IBM (phase 1 and 2) systems, symbolic links at the root level are available to the directories listed in Table 4 (for example /nwprod for phase 1 and /nwprod2 for phase 2). On the Cray XC40 systems, the se directories should be located using the variables defined in the *prod_envir* module (see Example 7 in Appendix A).

Table 4: WCOSS directory structure

Directory	**Description**
nwprod	applications/packages in the production suite
nwtest	applications/packages in the test suite (unscheduled)
nwpara	applications/packages in the parallel suite (scheduled)
nwbkup	backup of production packages (/nwprod)
nwges	model guess fields (spin-up data)
com	data and application output, including outgoing products
dcom	incoming data (retrieved from outside WCOSS)
pcom	outgoing products with WMO headers
tmpnwprd	temporary working directories for running jobs

Data from external sources is stored in **dcom** and model output is stored in **com**. The output folder of the com directory contains job stdout and stderr. Several forecast models produce model guess fields to be used as input for subsequent model runs. This spin-up data is stored in **nwges**. World Meteorological Organization (WMO) headed output products sent to the Telecommunication Operations Center (TOC) and onward to the Satellite Broadcast Network (SBN) are stored in **pcom**. Pcom data must be date-independent such that the data files are overwritten each day. Table 5 (below), Table 7, Table 8, and Table 9 (in Appendix B) show the structures of com, nwges, pcom and dcom directories, respectively.

Table 5: Structure of /com directory

Subdirectory	**Description**
NET/prod/*RUN.YYYYMMDD*	production model output for a day
NET/test/*RUN.YYYYMMDD*	test model output for a day
NET/para/*RUN.YYYYMMDD*	parallel model output for a day
output/prod/*YYYYMMDD*	production job stdout/stderror for a day
output/test/*YYYYMMDD*	test job stdout/stderror for a day
output/para/*YYYYMMDD*	parallel job stdout/stderror for a day
output/transfer/YYYYMMDD	transfer job stdout/stderror for a day
nawips/*envir*/*RUN.YYYYMMDD*	NAWIPS model output for a day
logs	log files

Appendix A: Workflow Examples

All examples are for job jpmb_forecast. Model name is nco and type of model run is pmb.

Example 5: Version file pmb.ver

The version file tracks the versions of all packages and modules used by your application.

`export pmb_ver=v1.1.0`	set the model version
`export nco_shared_ver=v1.0.6`	set the shared code version
`export grib_util_ver=1.0.1`	set the grib_util version

Example 6: Job card jpmb_forecast.job

In production, ecFlow preprocesses ecFlow scripts to generate job cards that are submitted to LSF. On the Cray XC40 system, production paths are set by loading the *prod_envir* module (Example 7). On the IBM system, they are exported individually within the job card (for production Phase 2 jobs, `NWROOT=/nwprod2`, `COMROOT=/com2`, `GESROOT=/nwges2`, and `PCOMROOT=/pcom2`). To read or write files from a development space, point the variables in your job card to the appropriate location(s).

`#BSUB -J jpmb_forecast_00`	job name
`#BSUB -o`	stdout/stderr
`/gpfs/hps/nco/ops/com/output/prod/today/pmb_forecast_00.o%J`	
`#BSUB -P PMB-OPS`	project identifier
`#BSUB -q prod`	LSF queue name
`#BSUB -L /bin/sh`	login shell
`#BSUB -W 00:30`	wall clock
`#BSUB -cwd /tmp`	protect your home directory
`#BSUB -M 100`	MAMU node memory alloc.
`#BSUB -extsched 'CRAYLINUX[]'`	schedule nodes via ALPS
`export NODES=8`	request 8 nodes
`%include <head.h>`	begin ecFlow communication
`export job=${job:-$LSB_JOBNAME}`	setup run environment for
`export jobid=${jobid:-$job.$LSB_JOBID}`	$job
`export RUN_ENVIR=${RUN_ENVIR:-nco}`	set $RUN_ENVIR to nco
`export envir=${envir:-prod}`	set $envir to prod
`export SENDDBN=${SENDDBN:-YES}`	alert files via DBNet
`export SENDDBN_NTC=${SENDDBN_NTC:-YES}`	alert AWIPS files via DBNet
`module load prod_util`	load production utilities
`module load prod_envir`	setup data root directories
`export jlogfile=${jlogfile:-`	set jlogfile location
`${COMROOT}/logs/jlogfiles/jlogfile.$jobid}`	
`export DATAROOT=${DATAROOT:-/gpfs/hps/nco/ops/tmpnwprd}`	set working dir. location
`export DBNROOT=/iodprod/dbnet_siphon`	set DBNet exec location
`export PCOMROOT=${PCOMROOT:-${PCOMROOT}/$envir}`	add $envir to pcom path
`export SENDECF=${SENDECF:-YES}`	send signals to ecFlow
`export SENDCOM=${SENDCOM:-YES}`	copy output files to com
`export KEEPDATA=${KEEPDATA:-NO}`	delete working dir. after run
`export cyc=00`	set the cycle
`export KMP_AFFINITY=disabled`	define parallel environment
`export MPICH_GNI_MAX_EAGER_MSG_SIZE=65536`	variables
`export FORT_BUFFERED=TRUE`	

`model=pmb`
`. ${NWROOT:?}/versions/${model}.ver`
	package name of *J*-job
	source version file

`module load grib_util/$grib_util_ver`	load grib utility module

`eval export HOME${model}=$NWROOT/$model.\$${model}_ver`
`eval \$HOME${model}/jobs/JPMB_FORECAST`
	define $HOMEpmb variable
	call *J*-job

`%include <tail.h>`	end ecFlow communication

Example 7: prod_envir module on Surge

To see what a module will do, run the "`module show`" or "`module display`" command.

```
> module show prod_envir
-------------------------------------------------------------------
module-whatis    Sets up variables for NCEP production suite paths
setenv           NWROOT      /gpfs/hps/nco/ops/nwprod
setenv           NWROOTp1    /gpfs/gp1/nco/ops/nwprod
setenv           NWROOTp2    /gpfs/gp2/nco/ops/nwprod
setenv           COMROOT     /gpfs/hps/nco/ops/com
setenv           COMROOTp1   /gpfs/gp1/nco/ops/com
setenv           COMROOTp2   /gpfs/gp2/nco/ops/com
setenv           GESROOT     /gpfs/hps/nco/ops/nwges
setenv           GESROOTp1   /gpfs/gp1/nco/ops/nwges
setenv           GESROOTp2   /gpfs/gp2/nco/ops/nwges
setenv           DCOMROOT    /gpfs/gp1/nco/ops/dcom
setenv           PCOMROOT    /gpfs/hps/nco/ops/pcom
-------------------------------------------------------------------
```

Example 8: *J*-job JPMB_FORECAST

`#!/bin/sh`	

`date`
`export PS4=' $SECONDS + '`
`set -x`
	print starting time
	prepend time to output
	enable verbose logging

`export DATA=${DATA:-${DATAROOT:?}/$jobid}`
`mkdir -p $DATA`
`cd $DATA`
	create temporary working directory

`export cycle=${cycle:-t${cyc}z}`
`setpdy.sh`
`. PDY`
	set up temporal variables, including PDY

`export SENDDBN=${SENDDBN:-YES}`
`export SENDECF=${SENDECF:-YES}`
	alert output via DBNet
	send signals to ecFlow

`export USHpmb=$HOMEpmb/ush`
`export EXECpmb=$HOMEpmb/exec`
`export PARMpmb=$HOMEpmb/parm`
`export FIXpmb=$HOMEpmb/fix`
	sub-directories of the current model

`export HOMEnco=${HOMEnco:-${NWROOT}/nco_shared.$nco_shared_ver}`
`export EXECnco=$HOMEnco/exec`
	provide access to nco shared executables

`export NET=${NET:-nco}`
`export RUN=${RUN:-pmb}`
	variables used in com directory organization

`export COMINgfs=${COMINgfs:-$(compath.py gfs/prod/gfs.$PDY)}`	locations of incoming data

`export getges_envir=${getges_envir:-prod}`
`export GESIN=${GESIN:-${GESROOT}/prod}`
`export COMIN=${COMIN:-$(compath.py ${NET}/${envir}/$RUN.$PDY)}`	

`export COMOUT=${COMOUT:-${COMROOT}/${NET}/${envir}/$RUN.$PDY}`
`export COMOUTarch=${COMOUTarch:-${COMROOT}/arch/$envir/syndat}`
`export PCOM=${PCOM:-${PCOMROOT}/$NET}`
`export GESOUT=${GESOUT:-${GESROOT}/$envir}`
	locations of outgoing data

`mkdir -p $COMOUT $PCOM $GESOUT`	create output directories
`export pgmout=OUTPUT.$$`	output for executables
`env`	print current environment

`$HOMEpmb/scripts/expmb_forecast.sh`
`export err=$?; err_chk`
	execute *ex*-script
	error checking

`postmsg $jlogfile "$0 completed normally"`	post successful completion message

`if [ -e "$pgmout" ]; then`
`  cat $pgmout`
`fi`
	print exec output

`if [ "${KEEPDATA^^}" != YES ]; then`
`  rm -rf $DATA`
`fi`
	remove temporary working directory

`date`	print ending time

Example 9: *ex*-script expmb_forecast.sh

`#!/bin/sh`	

`# Program Name: pmb_forecast`
`# Author(s)/Contact(s): First Last`
`# Abstract: Driver script for pmb forecast`
`# History Log:`
`#   5/2014: Added error checking`
`#   8/2014: Modified for WCOSS`
`#`
`# Usage:`
`#  Parameters: None`
`#  Input Files:`
`#    pmb.tHHz.anl`
`#  Output Files:`
`#    pmb.tHHz.fFFF.grib2`
`#`
`# Condition codes:`
`#    99  - Missing input file`
`#`
`# User controllable options: None`
	ex-script DOCBLOCK

`set -x`	enable verbose logging
`cpreq $COMIN/inputfile inputfile`	copy essential input files into working directory
`export pgm=pmb_forecast`	name of the binary executable

`. prep_step`
`export FORT11=$FIXpmb/inputfile.tbl`
`export FORT12=inputfile`
`export FORT60=outputfile.grib2`
	clear FORTRAN unit assignments
	set FORTRAN unit assignments

`startmsg`	log program start
`aprun -n 192 -N 24 $EXECmodel/$pgm >>$pgmout 2>errfile`	execute MPI program
`export err=$?; err_chk`	error checking
`if [ -s outputfile.grib2 ]; then`	check for required output
`  cpfs outputfile.grib2 $COMOUT/outputfile.grib2`	copy output file to output
`  if [ "${SENDDBN^^}" = YES ]; then`	directory
`    $DBNROOT/bin/dbn_alert MODEL PMB_FCST \`	alert output file
`        $job $COMOUT/outputfile.grib2`	
`  fi`	
`else`	
`  err_exit "outputfile.grib2 was not generated"`	terminate the job if the
`fi`	expected output cannot be
	found
`export pgm=tocgrib2`	Setup for tocgrib2 exec
`. prep_step`	
`export FORT11=outputfile.grib2`	define input file
`export FORT51=grib2.t${cyc}.z.pmb.f000`	define output file
`startmsg`	
`$TOCGRIB2 <$PARMpmb/grib2_awp_pmbf000 >>$pgmout 2>errfile`	add WMO header to file
`if [ $? -ne 0 ]; then`	error checking
`  msg="WARNING: WMO header not added to $FORT51"`	
`  postmsg $jlogfile "$msg"`	
`  echo "$msg" | mail.py`	
`fi`	

Example 10: build readme file sorc/README

```
Build instructions:
  1. cd to the sorc directory
  2. load the build_pmb module:
       module purge
       module use .
       module load build_pmb.module
  3. to build all executables:
       ./build_pmb.sh
     to build one or more executables, provide their name(s) as parameter(s):
       ./build_pmb.sh pmb_forecast pmb_post
```

Example 11: build script sorc/build_pmb.sh

`#!/bin/sh`	
`set -x`	enable verbose logging
`sorc_root=$PWD`	
`function build_dir {`	move to the source directory of the given executable
`  cd ${sorc_root}/$1`	make the executable
`  make`	if the build exited cleanly
`  if [ $? -eq 0 ]; then`	move the executable to the exec directory
`    make install`	clean the source directory
`    make clean`	
`  else`	print error message
`    echo "ERROR: build of $1 FAILED!"`	
`  fi`	exit the source directory
`}`	
`if [ $# -eq 0 ]; then`	if no parameters were given,
`  for source_dir in *.fd; do`	build all executables
`    build_dir $source_dir`	enter the build_dir function
`  done`	

`else`	if one or more executables were requested,
`  for source_dir in $*; do`	build those that were requested
`    build_dir $source_dir.fd`	enter the build_dir function
`  done`	
`fi`	

Example 12: modulefile sorc/build_pmb.module (to be loaded prior to compilation)

`#%Module######################################################`	module DOCBLOCK
`#                                      First.Last@noaa.gov`	
`#                                      ORGANIZATION`	
`# PMB-FCST v1.1.0`	
`##############################################################`	
`proc ModulesHelp { } {`	module help
`  puts stderr "Set environment variables for PMB-FCST"`	
`  puts stderr "This module initializes the user's"`	
`  puts stderr "environment to build the PMB model at NCEP"`	
`}`	
`module-whatis  "PMB-FCST whatis description"`	module description
`set    ver v1.1.0`	set version and
`setenv COMP intel`	compiler variables
`setenv FC ftn`	
	establish known
`# Known conflicts`	conflicts
`conflict PrgEnv-intel/5.2.40`	
`conflict NetCDF-intel-haswell/3.6.3`	
`conflict w3nco-intel/2.0.5`	
`# Load Cray parallel environment for Haswell architecture`	load ics and all ncep
`module load craype-haswell`	library modules used
	in the build process
`# Load Intel programming environment`	
`module load PrgEnv-intel/5.2.56`	
`# Load NCEP libs modules`	
`module load HDF5-serial-intel-haswell/1.8.9`	
`module load NetCDF-intel-haswell/4.2`	
`module load bacio-intel/2.0.1`	
`module load w3nco-intel/2.0.6`	
`module load jasper-gnu-haswell/1.900.1`	
`module load png-intel-haswell/1.2.49`	
`module load zlib-intel-haswell/1.2.7`	

Example 13: sorc/pmb_forecast.fd/makefile

`###############################################################`	Makefile DOCBLOCK
`#  Makefile for xxx`	containing
`#  Use:`	instructions and use
`#    make         - build the executable`	
`#    make install - move the built executable into the exec dir`	
`#    make clean   - start with a clean slate`	
`###############################################################`	
`# Tunable parameters:`	
`#   FC Name of the FORTRAN compiling system to use`	
`#   LDFLAGS Options of the loader`	
`#   FFLAGS Options of the compiler`	
`#   DEBUG Options of the compiler included for debugging`	
`#   LIBS List of libraries`	
`#   CMD Name of the executable`	

```
FC      = ${FC}   # Use Intel FORTRAN Compiler, ifort                  name of compiler
LDFLAGS = -O -convert big_endian                                       options of the loader
BINDIR  = ../../exec                                                   executable location
INC     = ${G2_INC4}                                                   include files
LIBS    = ${G2_LIB4} ${W3NCO_LIB4} ${BACIO_LIB4} ${JASPER_LIB}         libraries
${PNG_LIB} ${Z_LIB}
CMD     = pmb_forecast                                                 executable name
DEBUG   =                                                              debug options
FFLAGS  = -O3 -I $(INC) $(DEBUG)                                       compiler options

# Lines from here down should not need to be changed. They are
# the actual rules which make uses to build CMD.

all:  check_prereqs $(CMD)

$(CMD):     $(OBJS)
      $(FC) $(LDFLAGS) -o $(@) $(OBJS) $(LIBS)

clean:
      -rm -f $(OBJS) *.mod $(CMD)

install:
      -mv $(CMD) ${BINDIR}/

check_prereqs:                                                         check perquisite
      /gpfs/hps/nco/ops/nwprod/spa_util/check_libs.bash $(LIBS)         libraries and includes
      /gpfs/hps/nco/ops/nwprod/spa_util/check_incs.bash $(INC)
```

Appendix B: Variables and Directory Structure Tables

Table 6: Production utilities accessible via module variables

Variable	exec	Description
CNVGRIB	cnvgrib	Converts between GRIB1 and GRIB2
COPYGB	copygb	Copies all or part of GRIB1 file to another GRIB1 file
COPYGB2	copygb2	Copies all or part of GRIB2 file to another GRIB2 file
DEGRIB2	degrib2	Creates inventory of GRIB2 file
GRB2INDEX	grb2index	Creates index file from GRIB2 file
GRBINDEX	grbindex	Creates index file from GRIB1 file
GRIB2GRIB	grib2grib	Extracts GRIB records from a GRIB file made by gribawp1
TOCGRIB	tocgrib	Adds WMO header in front of each GRIB1 field
TOCGRIB2	tocgrib2	Adds WMO header in front of each GRIB2 field
TOCGRIB2SUPER	tocgrib2super	Adds WMO super header and time stamp to GRIB2 fields
WGRIB	wgrib	Creates inventory and decodes GRIB1 files
WGRIB2	wgrib2	Creates inventory and decodes GRIB2 files
NDATE	ndate	Date utility
MDATE	mdate	Date utility
NHOUR	nhour	Date utility
FSYNC	fsync_file	Synchronize file across GPFS

Table 7: Structure of /nwges directory

Subdirectory	Description
prod/*model.YYYYMMDD*	production spin-up data for model

test/*model.YYYYMMDD*	test spin-up data
para/*model.YYYYMMDD*	parallel spin-up data

Table 8: Structure of /pcom directory

Subdirectory	Description
prod/*model*	production WMO headed output products
test/*model*	test WMO headed output products
para/*model*	parallel WMO headed output products

Table 9: Structure of /dcom directory

Subdirectory	Description
us007003/*YYYYMMDD*	incoming data for one day
us007003/*YYYYMM*	Incoming data for one month (select types only)
us007003/*YYYYMMDD*/b*TTT*/xx*SSS*	data tanks

TTT and *SSS* correspond to the 3-digit BUFR data category type and sub-type, respectively