NCEP Central Operations

WCOSS Implementation Standards

May 13, 2015

Version 10.0

I. Introduction

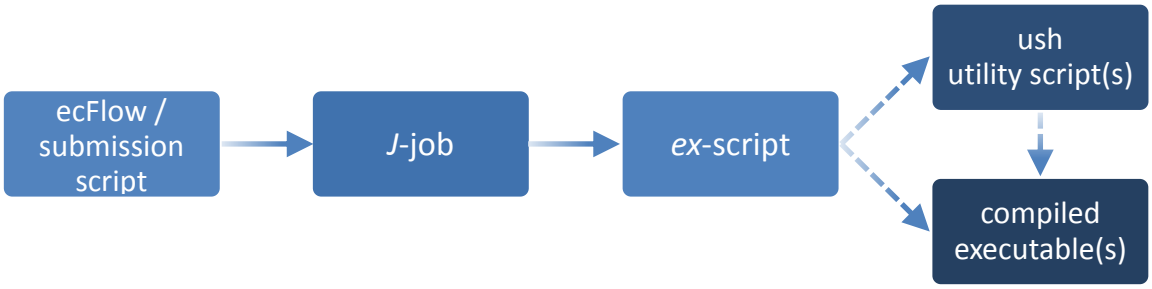
The reliable production and availability of the National Center for Environmental Prediction's (NCEP) guidance products plays a critical role in the mission of the National Weather Service to provide forecasts and warnings "for the protection of life and property and the enhancement of the national economy." 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) Production Management Branch (PMB). 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 acts like an LSF submission script, setting up the *bsub* parameters and much of the execution environment and calling the *J*-job to execute the job. All jobs must be submitted to LSF via *bsub*. It is at the ecFlow (NCO) or submission script level (development organizations) where certain environment specific variables must be set (See Section III-A for further details).

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 utility (*ush*) scripts. Additional discussion and examples of the expected 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 configuration script or *J*-Job. For example, to move a model from development to production, it should only be necessary to change the exported variables in the ecFlow scripts / configuration scripts or *J*-Jobs. 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. They must be used wherever appropriate. In the production environment, several of these variables are set in ecFlow. Developers should set these variables in a separate wrapper or LSF submission script in order to keep the *J*-jobs as clean as possible. Table 1 delineates standard environment variables and where they are typically set in the production workflow. If any of the below variables are not used in a given job then do not set them in the *J*-job.

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.	ecFlow*
envir	Set to "test" during the initial testing phase, "para" when running in parallel (on a schedule), and "prod" in production.	ecFlow*
NWROOT	Root directory for application, typically /nw`$envir`	ecFlow*
NWROOT*system*	Application root directory on alternate system (*i.e.* `$NWROOTp1`)	ecFlow*
job	Unique job name (unique per day and environment)	ecFlow*
jobid	Unique job identifier, typically `$job.$$` (where `$$` is an ID number)	ecFlow*
jlogfile	Log file for start time, end time, and error messages of all jobs	ecFlow*
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
PDYm1-7	Dates of previous seven days in YYYYMMDD format (`$PDYm1` is yesterday's date, etc.)	*J*-job
PDYp1-7	Dates of next seven days in YYYYMMDD format (`$PDYp1` is tomorrow's date, etc.)	*J*-job
cyc	Cycle time in GMT, formatted HH	ecFlow
cycle	Cycle time in GMT, formatted tHHz	*J*-job
DATAROOT	Directory containing the working directory, often /tmpnwprd1 in production	ecFlow*
DATA	Location of the job working directory, typically `$DATAROOT/$jobid`	*J*-job
HOME*model*	Application home directory, typically `$NWROOT/`*`model`*`.v`*`X.Y.Z`*	ecFlow

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	ecFlow*
DCOM	dcom directory for model input data	*J*-job
COMROOT	com root directory for input/output data on current system	ecFlow*
COMROOT*system*	com root directory for input/output data on alternate system (*i.e.* $COMROOTp1)	ecFlow*
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	ecFlow*
GESROOT*system*	nwges root directory for input/output guess fields on alternate system (*i.e.* $GESROOTp1)	ecFlow*
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	ecFlow*
PCOM	pcom directory for outgoing products with WMO headers; typically $PCOMROOT/$NET	*J*-job
DBNROOT	Root directory for the data alerting utilities	ecFlow*
SENDCOM	Boolean† variable to control data copies to $COMOUT	ecFlow*
SENDECF	Boolean† variable used to control ecflow_client child commands	ecFlow*
SENDDBN	Boolean† variable used to control sending products off WCOSS	ecFlow*
SENDDBN_NTC	Boolean† variable used to control sending products with WMO headers off WCOSS	ecFlow*
SENDWEB	Boolean† variable used to control sending products to a web server, often ncorzdm	ecFlow*
***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
KEEPDATA	Boolean† variable used to specify whether or not the working directory should be deleted upon successful job completion.	ecFlow*

*variables set in envir.h ecFlow header (see Example 6); should be defined in development wrapper script

†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. 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 5 (in Appendix B) for variables and their descriptions. The module must be loaded in all production jobs by calling "`module load prod_util`." 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 `$err` variable following the execution of a program. If `$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
```

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. As with above, access to the below date utilities is done 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=20150101

# 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 20150101
The date ten days ago was 20141222
The date in ten days will be 20150111
The last four days were 20141231 20141230 20141229 20141228
The next four days are 20150102 20150103 20150104 20150105
```

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 ahead is $next_cycle"
echo "Six hours previous is $prev_cycle"
```

Output
```
Today's date and cycle is 2014112615
```

```
The date ten days ago was 2014111615
The date in ten days will be 2014120615
Six hours ahead is 2014112621
Six hours previous is 2014112609
```

setpdy.sh

`setpdy.sh` creates a file PDY that is sourced to export the standard date variables PDYm1-7, PDY, and PDYp1-7. The variable `cycle` must be set (in 't*HH*z' 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 by the prod_setup job run 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 20150101)

Script

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

setpdy.sh
. PDY

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

Contents of file PDY

```
export PDYm7=20141225
export PDYm6=20141226
export PDYm5=20141227
export PDYm4=20141228
export PDYm3=20141229
export PDYm2=20141230
export PDYm1=20141231
export PDY=20150101
export PDYp1=20150102
export PDYp2=20150103
export PDYp3=20150104
export PDYp4=20150105
export PDYp5=20150106
export PDYp6=20150107
export PDYp7=20150108
```

Output

```
Yesterday's date was 20141231
```

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 5 (in Appendix B) for all variable definitions and descriptions of each utility. The module must be loaded at the *J*-job level of all jobs using GRIB utilities.

```
module load grib_util
```

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 that 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 in the script to alert (*e.g.* e-mail) the appropriate parties when primary data is unavailable. Note that e-mail notifications can only be sent from jobs running on the prod_serv nodes.

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 (*i.e.* links to absolute paths) are not allowed in application directories. When external resources are required, external paths should be 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 can not 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: suggested 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 (#, !) 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 on WCOSS or higher (icc and ifort).
3. All libraries must be approved for production use. Makefiles should include compilers and libraries using variables defined in the associated module and described in README, for example:

   command line (README):

   ```
   module load w3nco/v2.03
   module load ics/v15.0.1
   ```

   makefile:

   ```
   LIBS = ${W3NCO_LIB4}
   ndate: ndate.f
          ifort –o ndate ndate.f $(LIBS)
   ```

Modulefiles should be used for more complex builds. See Example 9 in Appendix A for an example modulefile.

4. In order for certain errors to be trapped early in the build process, it is recommended to add the check_prereqs target to all makefiles:

```
check_prereqs:
      /nwprod/spa_util/check_libs.bash $(LIBS)
      /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 must be able to compile without any 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 and mark subsequent variable definitions for export at the top of each *J*-job:

```
export PS4=' $SECONDS + '
set -xa
```

2. Utilize standard environment variables and utilities (See Section III-A).
3. Each block of copies from the scratch directory to **com**, **nwges** or **pcom** must be wrapped with logic testing if the variable $SENDCOM is set to "YES". Never write to **dcom**!
4. Each block of dbnet alerts must be wrapped with logic testing whether the variable $SENDDBN or $SENDDBN_NTC, as applicable, is set to "YES".
5. Each execution of a C or Fortran code must be wrapped with the production utilities `prep_step`, `startmsg` and `err_chk`.
6. Each execution should redirect standard out to `$pgmout` and standard error to *errfile*:

```
$EXECmodel/$pgm >> $pgmout 2> errfile
```

7. 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.
8. 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.
9. Remove all references to developer work areas and all development tools (benchmarking, etc.) before submitting to PMB.
10. 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`.

Reference Appendix A for examples of an ecFlow submit script, environment configuration script, *J*-job, *ex*-script, modulefile and makefile with notes explaining the purpose of different sections.

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 PMB'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 any pre-built executables or libraries to PMB. It is the SPA's responsibility to build all executables and libraries before running an application on WCOSS.
3. If more than one executable is to be built, divide the source files into sub-directories according to the executable they produce. 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). In this way, a simple "build all" script can be written to batch process the building of all executables for a given application.
4. Any application containing source code should be delivered with a module file, which will be used to set up the environment to build all executables within the delivered package. A modulefile will allow PMB to keep track of the compiler, library versions, and any other external files used to compile the application. An example modulefile can be found in Example 9 of Appendix A. Creating symbolic links to external resources (*i.e.* to absolute paths) is not allowed.
5. It is preferable for each source code directory to have a makefile that does everything needed to build one executable. For example, global_fcst.fd would contain Fortran code and a makefile to produce the global_fcst executable. An example makefile can be found in Example 10 of Appendix A .
6. Use a readme file in the source directory to explain the build process, especially if it requires any interaction or if it is non-standard in any way. This includes information on any situations where a makefile produces more than one executable. An explanation of how to build in the same directory as the source will eliminate confusion and reduce errors if it becomes necessary to rebuild the executable to resolve a production failure or other emergency situation.

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 directory. The application directory must contain all *J*-jobs and *ex*-scripts specific to a given 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 2). If there exists code, scripts or other files shared between multiple models then they should reside in a shared directory under /nwprod with the following naming convention: *model*/*function*_shared.v*X.Y.Z* (*e.g.* gsi_shared.v5.0.0). The shared directory should never contain a jobs sub-directory.

Table 2: Application 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 or other static input data
fix	fixed fields, tables or other static input data
lib	model-specific libraries
ecf	ecFlow / submission scripts and ecFlow definition files (developers not responsible for this directory)
gempak	all gempak related files

Table 3 lists the primary data and application directories used within the WCOSS NCO production environment. 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 so the data stored will overwritten each day. Table 4 (below), Table 6, Table 7, and Table 8 (in Appendix B) show the structures of com, nwges, pcom and dcom directories, respectively.

Table 3: WCOSS Directory Structure

Directory	Description
/nwprod	applications in the production suite
/nwtest	applications in the test suite (unscheduled)
/nwpara	applications in the parallel suite (scheduled)
/nwbkup	backup of production applications (/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 (separate directory for each filesystem)

Table 4: 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: ecFlow script jpmb_forecast.ecf

`#BSUB -J %E%pmb_forecast_%CYC%`	job name
`#BSUB -o /com/output/%ENVIR%/today/pmb_forecast_%CYC%.o%J`	stdout/stderr
`#BSUB -P %PROJ%`	project identifier
`#BSUB -q %QUEUE%`	LSF queue name
`#BSUB -L /bin/sh`	login shell
`#BSUB -W 00:30`	wall clock
`#BSUB -cwd /tmpnwprd`	initial working directory
`#BSUB -x`	use exclusive nodes
`#BSUB -n 8`	number of tasks
`#BSUB -R span[ptile=8]`	number of tasks per node
`#BSUB -R affinity[core]`	affinitize to 1 core
`#BSUB -R rusage[mem=5000]`	use 5000MB of memory
`#BSUB -a poe`	use poe
`%include <head.h>`	ecFlow headers
`%include <envir.h>`	setup NCO environment
`module load grib_util`	load grib utility module
`export cyc=%CYC%`	set the cycle
`export model=pmb`	base name of model directory
`export MP_LABELIO=YES`	define all poe variables here, if applicable
`export MP_USE_BULK_XFER=YES`	
`export MP_PULSE=500`	
`VERSION_FILE=${NWROOT:?}/versions/$model.ver`	source model version file
`if [ -f $VERSION_FILE ]; then`	
`. $VERSION_FILE`	
`else`	
`ecflow_client --abort="$VERSION_FILE does not exist"`	send ecFlow abort signal if the version file does not exist
`exit`	
`fi`	
`${NWROOT}/${model}.${pmb_ver}/jobs/JPMB_FORECAST`	call J-Job
`%include <tail.h>`	ecFlow footer
`%manual`	manual section that should include a brief summary of the job's purpose and troubleshooting tips
`%end`	

Example 6: Environment configuration script (ecFlow envir.h)*

* An example configuration script that may differ from the actual envir.h used in production

`export job=${job:-$LSB_JOBNAME}`	setup run environment for $job
`export jobid=${jobid:-$job.$LSB_JID}`	
`export RUN_ENVIR=${RUN_ENVIR:-nco}`	
`export envir=%ENVIR%`	
`module load prod_util`	load prod utility module
`export DCOMROOT=${DCOMROOT:-/dcom/us007003}`	setup data root directories
`export COMROOT=${COMROOT:-/com}`	
`export GESROOT=${GESROOT:-/nwges}`	

```
case $envir in
  prod)
    export jlogfile=${jlogfile:-
           ${COMROOT}/logs/jlogfiles/jlogfile.${jobid}}
    export DATAROOT=${DATAROOT:-/tmpnwprd1}
    export DBNROOT=/iodprod/dbnet_siphon
    ;;
  eval)
    export envir=para
    export jlogfile=${jlogfile:-
                     ${COMROOT}/logs/${envir}/jlogfile}
    export DATAROOT=${DATAROOT:-/tmpnwprd2}
    export DBNROOT=/nwprod/spa_util/para_dbn
    SENDDBN_NTC=NO
    ;;
  para|test)
    export jlogfile=${jlogfile:-
                     ${COMROOT}/logs/${envir}/jlogfile}
    export DATAROOT=${DATAROOT:-/tmpnwprd2}
    export DBNROOT=/nwprod/spa_util/fakedbn
    KEEPDATA=YES
    ;;
esac

export NWROOT=${NWROOT:-/nw${envir}}
export PCOMROOT=${PCOMROOT:-/pcom/${envir}}
export SENDDBN=${SENDDBN:-YES}
export SENDDBN_NTC=${SENDDBN_NTC:-YES}
export SENDWEB=${SENDWEB:-YES}
export SENDECF=${SENDECF:-YES}
export SENDCOM=${SENDCOM:-YES}
export KEEPDATA=${KEEPDATA:-NO}
```

setup variables specific to the NCO production environment

setup variables specific to the NCO parallel evaluation environment

setup variables specific to the NCO parallel and test environments

setup environment variables common to all NCO environments

clean up $DATA

Example 7: *J*-job JPMB_FORECAST

All variables defined in the *J*-job should default to NCO production environment

```
#!/bin/sh

date
export PS4=' $SECONDS + '
set -x

export DATA=${DATA:-${DATAROOT:?}/$jobid}
mkdir -p $DATA
cd $DATA

export cycle=${cycle:-t${cyc}z}
setpdy.sh
. PDY

export SENDCOM=${SENDCOM:-YES}
export SENDDBN=${SENDDBN:-YES}
export SENDECF=${SENDECF:-YES}

export HOMEpmb=${HOMEpmb:-${NWROOT:?}/pmb.$pmb_ver}
export USHpmb=$HOMEpmb/ush
export EXECpmb=$HOMEpmb/exec
export PARMpmb=$HOMEpmb/parm
export FIXpmb=$HOMEpmb/fix
```

print starting time
prepend time to output
enable verbose logging

create temporary working directory

set up temporal variables, including PDY

send output to COM
alert output through DBNet
send signals to ecFlow

parent directory and all sub-directories for 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=${COMINgfs:-${COMROOT}/gfs/prod/gfs.$PDY}`
`export getges_envir=${getges_envir:-prod}`
`export GESIN=${GESIN:-${GESROOT}/prod}`
`export COMIN=${COMIN:-${COMROOT}/${NET}/${envir}/$RUN.$PDY}`
	locations of incoming data

`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

`if [ "$SENDCOM" = YES ]; then`
`  mkdir -p $COMOUT $PCOM $GESOUT`
`fi`
	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
`msg="JOB $job HAS COMPLETED NORMALLY."` `postmsg $jlogfile "$msg"`	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 8: *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
`pgm=pmb_forecast`	name of the binary executable
`. prep_step`	clear Fortran unit assignments

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

`startmsg`
`$EXECmodel/$pgm >> $pgmout 2> errfile`
`export err=$?; err_chk`
	log program start
	execute program
	error checking

`if [ -s outputfile.grib2 ]; then`
`  if [ "$SENDCOM" = YES ]; then`
`  cpfs outputfile.grib2 $COMOUT/outputfile.grib2`
`  if [ "$SENDDBN" = YES ]; then`
`  $DBNROOT/bin/dbn_alert MODEL PMB_FCST \`
`  $job $COMOUT/outputfile.grib2`
`  fi`
`  fi`
	check for required output
	
	copy output file to output directory
	alert output file

`else`
`  err_exit "outputfile.grib2 was not generated"`
`fi`
	terminate the job if the expected output cannot be found

`pgm=tocgrib2` `. prep_step`	Setup for tocgrib2 exec
`export FORT11=outputfile.grib2` `export FORT51=grib2.t${cyc}.z.pmb.f000`	define input file define output file

`startmsg`
`$TOCGRIB2 <$PARMpmb/grib2_awp_pmbf000 >>$pgmout 2>errfile`
`if [ $? -ne 0 ]; then`
`  msg="WARNING: WMO header not added to $FORT11"`
`  postmsg $jlogfile "$msg"`
`  echo "$msg"`
`fi`
	add WMO header to file
	error checking

Example 9: modulefile PMB

This example represents a model's modulefile. It should be loaded before compiling the source code for the application. A similar modulefile can be created for libraries.

`#%Module#####################################################`
`######################################################################`
`##  First.Last@noaa.gov`
`##  *ORGANIZATION*`
`##  PMB-FCST v1.0.0`
`######################################################################`
	module DOCBLOCK

`proc ModulesHelp { } {`
`  puts stderr "Set environment variables for PMB-FCST"`
`  puts stderr "This module initializes the users"`
`  puts stderr "environment to build the PMB model at NCEP"`
`}`
	module help

`module-whatis  "PMB-FCST whatis description"`	module description
`set  ver v1.0.0` `setenv COMP ifort`	set version and compiler variables

```
# Known conflicts                                          establish known
conflict ics/12.1                                          conflicts
conflict w3nco/v2.0.5
conflict w3nco/v2.0.4

#Loading intel suite
module load ics/15.0.1                                     load ics and all ncep

# Loding ncep libs modules                                 library modules used
module load EnvVars/1.0.0                                  in the build process
module load HDF5/1.8.9/serial
module load NetCDF/3.6.3
module load bacio/v2.0.1
module load w3nco/v2.0.6
module load jasper/v1.900.1
module load png/v1.2.44
module load z/v1.2.6
```

Example 10: pmb_forecast.fd/makefile

```
############################################################## Makefile DOCBLOCK
#    Makefile for xxx                                        containing
#    Use:                                                    instructions and use
#      make         -  build the executable
#      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      = ${COMP}   # 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
      /nwprod/spa_util/check_libs.bash $(LIBS)             libraries and includes
      /nwprod/spa_util/check_incs.bash $(INC)
```

Appendix B: Variables and Directory Structure Tables

Table 5: Production variable definitions accessible by modules

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 6: 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 7: 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 8: 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