# NCEP Central Operations

# WCOSS Implementation Standards

April 17, 2019

Version 10.2.0

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 four distinct phases, referred to in this document as IBM Phase 1, IBM Phase 2, Cray XC40, and Dell Phase 3.  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 must only be necessary to change the variables exported in the job cards.  Downstream scripts must always use the variables established in the *J*-job and must never alter them.
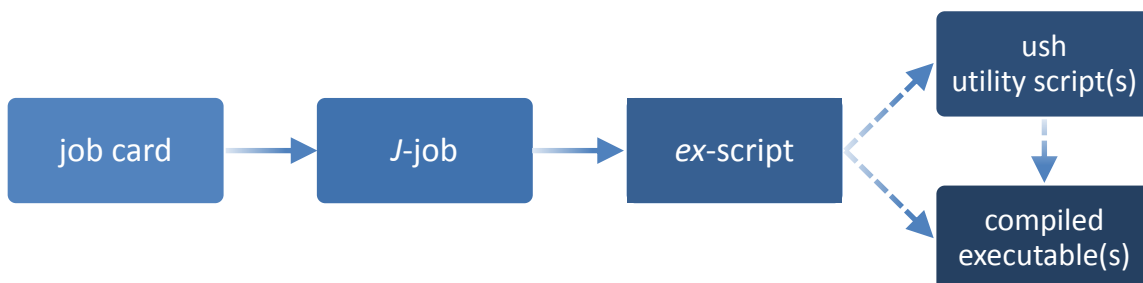
# III. Standard Variables, Formats, and Utilities

## 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 defined in the job card generated by ecFlow.  On the Cray XC40 and Dell Phase 3 systems, 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, e.g. /nw$envir | job card |
| **NWROOT*system*** | Application root directory on alternate system (*e.g.* $NWROOTp1), where *system* is one of p1, p2, xc40 or p3 | job card |
| **job** | Unique job name (unique per cycle and environment) | job card |
| **jobid** | Unique job identifier, typically $job.$LSB_JOBNAME | job card |
| **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 ($PDYm1 is yesterday's date, i.e. "present day minus 1", etc.) | *J*-job |
| **PDYp#** | Dates of a future day in YYYYMMDD format ($PDYp1 is tomorrow's date, i.e. "present day plus 1", etc.) | *J*-job |
| **cyc** | Cycle time in GMT hours, formatted HH | job card |
| **cycle** | Cycle time in GMT, formatted tHHz or tHHMMz | *J*-job |
| **subcyc** | Cycle time in GMT minutes, formatted MM | job card |
| **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 data ingested from remote sources | *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 (*e.g.* $COMROOThps for cray data and $COMROOTp3 for phase 3) | 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 (*e.g.* $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 |
| **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 is used at runtime by model *model* | version file |
| **KEEPDATA** | Boolean† variable used to specify whether or not the working directory must 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 must also be used.  File names must not contain special characters, uppercase characters or the date (the directory in which the file resides will contain the date).  File names must 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 must be consistent across environments and application versions, so variables such as $job, $envir, and $*model*_ver must 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 the wmo sub-directory:

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

Example filenames for files in the wmo sub-directory:

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 must be communicated as descriptively as possible and prefaced with "WARNING:" or "FATAL ERROR:".  Failures must not be allowed to propagate downstream of the point where the problem can first be detected.  The following utilities must 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 common production utility scripts 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 Intel 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.

*startmsg and postmsg will no longer be required in operations but the utilities will continue to be maintained. There will no longer be a central jlogfile. Instead, these utilities will direct messages to stdout or to any application specific location, as defined by the jlogfile variable

### 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 errfile and `$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
```

Recent versions of `cpfs` call the `err_exit` utility if either the cp or mv step returns non-zero status. However, this behavior may change.  To cover all bases, check whether the source file exists before calling cpfs.  If the job should continue without the file, skip the cpfs call and continue.  If the job should fail if the source file does not exist, call err_exit directly.  Either way, the exit status from cpfs must be checked and a warning printed if the status is non-zero.

### 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 standard `mail` command is not available on compute nodes so 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**

| System | *no flags* | –p | –t | –tp |
|---|---|---|---|---|
| Tide phase 1 | Tide | Tide-p1 | IBM | IBM-p1 |
| Tide phase 2 | Tide | Tide-p2 | IBM | IBM-p2 |
| Luna | Luna | Luna-XC40 | Cray | Cray-XC40 |
| Mars | Mars | Mars-p3 | Dell | Dell-p3 |
| Gyre phase 1 | Gyre | Gyre-p1 | IBM | IBM-p1 |
| Gyre phase 2 | Gyre | Gyre-p2 | IBM | IBM-p2 |
| Surge | Surge | Surge-XC40 | Cray | Cray-XC40 |
| Venus | Venus | Venus-p3 | Dell | Dell-p3 |

## 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 PDYm$n_m$, PDYm{$n_m$–1}, PDYm{$n_m$–2}, …, PDYm2, PDYm1, PDY, PDYp1, PDYp2, …, PDYp{$n_p$–2}, PDYp{$n_p$–1}, PDYp$n_p$. 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 $COMDATEROOT/date/$`cycle`, but it can be overridden by setting the variable PDY prior to execution. The date files in $COMDATEROOT/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 *(unless variable PDY was imported)* 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 that end, all code must 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 must 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. 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 must be logged indicating that the job is continuing without this data source (similar to use of backup data above).

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

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

v.  Minimize the time it takes to re-run a failed job
    Restart capability must be applied to save time when recovering from a failure. 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. Any job that runs longer than 15 minutes is required to have restart capability built in such that the process picks up where it left off when rerun.

vi. No background processing
    LSF loses control of processes when they are put in the background. Therefore, background processing must be avoided.

vii. No external-pointing symlinks
    Symbolic links to resources outside of the *application directory* or *package* (*e.g.* links to absolute paths) are not allowed within the package. When external resources are required, their paths must be obtained from production module variables (when available) or defined as variables in the version file and ecf script and used wherever the external resource is needed.

viii.   Working directories

Working directories must contain a unique identifier (job id) 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 must be removed upon successful completion of the run. All data that is needed for longer than one cycle must be copied to $COMOUT or $GESOUT. MPMD child processes must do their work in separate subdirectories of the main working directory to avoid cases where multiple processes might create/modify/remove the same file simultaneously.

ix.   Documentation Blocks

Source code and scripts must 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).

x.   Points of contact

All applications running in production must have a primary and backup support contact reachable 24/7 in case of operational failures.

**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 or on Dell Phase 3 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 must 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 all builds. See Example 11, Example 12, and Example 13 in Appendix A for an example build script, modulefile, and makefile, respectively.

4. Do not specify absolute paths to executables, libraries, or any other products inside the makefile.

5. Code must compile without errors and warnings.

6. Errors must be caught as early as possible and the context of the error must be communicated clearly. Failures must not be allowed to propagate past the point where the problem is first detectable. "Missing GFS data" is not an adequate error message. Indicate the specific GFS file and directory that is missing in the error message.

7. Input/output errors must be handled gracefully.  See available I/O control options to trap errors and add logic to allow the code to continue or fail as appropriate.

8. When an executable aborts, has other problems or needs to be tested, it is vitally important to know which disk files it uses for input and output To accomplish this, the following is required:

a) File names must not be hard-coded in the source code but rather defined in the calling script.  This can be done in one of the following ways:

- By using FILE=var option in the OPEN statement, where var is a character variable; the variable value must be exported to the shell environment before calling the executable and retrieved from the environment by either the routine GETENV (Fortran extension, requires "use IFPORT" in ifort) or the Fortran-2003 standard intrinsic GET_ENVIRONMENT_VARIABLE.
- (An ifort extension) by omitting the FILE= option, in which case the file name must be set by exporting the value of the character FORTn variable, where n is the Fortran I/O unit number as set in the OPEN statement.  For ifort, n is any positive integer fitting in a 4-byte variable. The production utility "prep_step" (clearing the values of all FORTn variables) must be called before each executable if this method is used.
- By omitting the FILE=var option, and not setting the FORTn variable, in which case the default file name "fort.n" will be used by the executable.  This method is allowed only if this file is a symbolic link, eg:  ln -sf $DATA/pgrbf01 fort.11.

b) It must be clear, by looking at the file names defined before calling the executable,  which files are read from (input), written to (output), and which are both read and written within the same executable (work files).  It can be ensured by one of the following:

- Using numbers 11-49 for input, 51-79 for output, 80-94 for work files (preferred method for executables opening a small number of files).
- Exporting separately the three groups of file names with appropriate headers / comments at the top of each block.

9. Good programming practices must be followed to improve readability.  Eg, structured control must be used instead of GO TO statements and code must be well documented.


## 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 must 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 must be written in Bash, Korn shell, Perl, or Python.  *Ex*-scripts must 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 must 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`, if applicable, and `err_chk`.
5. Any executions that print verbose output (more than 100 lines or so per execution) must redirect standard out to $pgmout and standard error to *errfile*:
   ```
   $EXECmodel/$pgm >> $pgmout 2> errfile
   ```
6. Production utilizes a centralized cleanup of directories in COMROOT and GESROOT. Production scripts must not remove directories at the $COMROOT/$NET/$envir/$RUN.$PDY level. Output must conform to the output structure of $COMROOT/$NET/$envir/$RUN.$PDY.
7. Do not assume that the current directory (".") will be in the execution path ($PATH).  *(Invoke temporary script as ./scriptx or $DATA/scriptx).*
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 must 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.
10. Never write to **dcom**! Unless you run data ingest from an outside source.

11. Ensure that files containing restricted data are assigned the appropriate group and permissions.
12. There must be no false/misleading errors and no syntax errors in the standard output/error file.
13. Ensure all non-zero stops, aborts, calls to err_exit, etc are for good reason. (Eg, consider whether a bad observation should be skipped rather than causing the job to fail).

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 or any other version control system that WCOSS has access to. 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 must 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. The only exception is if multiple executables share a large

portion of their code base in which case sub-directory sharing is allowed. The name of each source directory must be the name of the executable it produces plus the appropriate extension (.cd or .fd for C or FORTRAN code, respectively). If multiple executables are produced then their names must resemble the base source directory name.

4. All source code must be delivered with a modulefile and/or build script used to set up the build environment. It must 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 (*e.g.* to absolute paths) is not allowed.

5. Each source code directory must 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 must not move the compiled binary; however, 'make install' may do so.  Example 13 of Appendix A contains an example. See Environment Equivalence (EE) standards for more details about builds

6. There are five critical targets that must be defined in every makefile. They are all, debug, install, clean and test. Example 13 of Appendix A contains an example of each.

7. 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 must be named with the following format: *model.vX.Y.Z* (*e.g.* gfs.v12.0.1). Files must 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 must reside in a separate shared package (*e.g.* *model*_shared.v5.0.0). Shared packages must not contain *J*-jobs or a jobs sub-directory. Shared packages must be backward compatible.

### 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 and Dell Phase 3 systems, the same directories can 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) |
| 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 are placed in com under a **wmo** sub-directory.  Table 5 (below), Table 7, Table 8, and Table 9 (in Appendix B) show the structures of com, nwges, and dcom directories, respectively.

**Table 5: Structure of COMROOT 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 |
| logs | log files |

## C. Unresolved Bugs

Before handing off code to NCO, all Bugzilla entries must be addressed.  Please mark all items that have been resolved as such and add a brief complete explanation of the resolution, including relevant files modified to address the bug.  The SPA will then verify the fix during testing and close the bug following

implementation.  If a bug cannot be resolved, a comment must be added and approval received from the SPA team lead.

# 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.ecf

In production, ecFlow preprocesses ecFlow scripts to generate job cards that are submitted to LSF.  On WCOSS, production paths are set by loading the *prod_envir* module (Example 7).  To read or write files from a development space, point the variables in your job card to the appropriate location(s).

| | |
|---|---|
| `#BSUB –J %E%pmb_forecast_00` | job name |
| `#BSUB –o %COM%/output/prod/today/pmb_forecast_00.o%J` | stdout/stderr |
| `#BSUB -P %PROJ%-%PROJENVIR%` | project identifier |
| `#BSUB -q %QUEUE%` | LSF queue name |
| `#BSUB –L /bin/sh` | login shell |
| `#BSUB –W 00:30` | wall clock |
| `#BSUB –cwd $temp_space` | protect your home directory |
| `#BSUB –M 100` | MAMU node memory alloc. |
| `#BSUB -extsched 'CRAYLINUX[]'` | schedule nodes via ALPS |
| `export NODES=8` | request 8 nodes |
| | |
| | begin ecFlow communication |
| `%include <head.h>` | set up environment |
| `%include <envir-xc40.h>` | |
| | |
| | set model name |
| `export model=pmb` | source version file |
| `%include <model_ver>` | set the cycle |
| `export cyc=%CYC%` | |
| | |
| `export KMP_AFFINITY=disabled` | define parallel environment |
| `export MPICH_GNI_MAX_EAGER_MSG_SIZE=65536` | variables |
| `export FORT_BUFFERED=TRUE` | |
| | |
| `module load util_shared/$util_shared_ver` | |
| `module load grib_util/$grib_util_ver` | load only modules need for this job |
| | |
| `$HOMEmodel/jobs/JPMB_FORECAST` | |
| | call *J*-job |
| `%include <tail.h>` | |
| | end ecFlow communication |

Note that the envir-*phase*.h include files set the following environment variables in addition to loading the *prod_envir* and *prod_util* modules:

- job
- jobid
- RUN_ENVIR
- envir

- SENDDBN
- SENDDBN_NTC
- DATAROOT
- DBNROOT
- COREROOT
- SENDECF
- SENDCOM
- KEEPDATA

### 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            NWROOTp3    /gpfs/dell1/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            COMROOTp3   /gpfs/dell1/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            GESROOTp3   /gpfs/dell1/nco/ops/nwges
setenv            DCOMROOT    /gpfs/gp1/nco/ops/dcom
------------------------------------------------------------------------
```

### Example 8: *J*-job JPMB_FORECAST

```
#!/bin/sh

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

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

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

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

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


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

export COMINgfs=${COMINgfs:-$(compath.py gfs/prod/gfs.$PDY)}   locations of incoming data
export GESIN=${GESIN:-${GESROOT}/prod}
export COMIN=${COMIN:-${COMROOT}/${NET}/${envir}/$RUN.$PDY}

export COMOUT=${COMOUT:-${COMROOT}/${NET}/${envir}/$RUN.$PDY}
export COMOUTwmo=${COMOUTwmo:-${COMOUT}/wmo}          locations of outgoing data
export COMOUTgempak=${COMOUTwmo:-${COMOUT}/gempak}
export GESOUT=${GESOUT:-${GESROOT}/$envir}
```

| | |
|---|---|
| ```mkdir –p $COMOUT $COMOUTgempak $COMOUTwmo $GESOUT``` | create output directories |
| ```export pgmout=OUTPUT.$$``` | output for executables |
| ```env``` | print current environment |
| ```$HOMEpmb/scripts/expmb_forecast.sh```<br>```export err=$?; err_chk``` | execute *ex*-script<br>error checking |
| ```postmsg $jlogfile "$0 completed normally"``` | post successful completion<br>message |
| ```if [ -e "$pgmout" ]; then```<br>```  cat $pgmout```<br>```fi``` | print exec output |
| ```if [ "${KEEPDATA^^}" != YES ]; then```<br>```  rm –rf $DATA```<br>```fi``` | remove temporary<br>working directory |
| ```date``` | print ending time |

### Example 9: *ex*-script expmb_forecast.sh

| | |
|---|---|
| ```#!/bin/sh``` | |
| ```# Program Name: pmb_forecast```<br>```# Author(s)/Contact(s): First Last```<br>```# Abstract: Driver script for pmb forecast```<br>```# History Log:```<br>```#   5/2014: Added error checking```<br>```#   8/2014: Modified for WCOSS```<br>```#```<br>```# Usage:```<br>```#  Parameters: None```<br>```#  Input Files:```<br>```#    pmb.tHHz.anl```<br>```#  Output Files:```<br>```#    pmb.tHHz.fFFF.grib2```<br>```#```<br>```# Condition codes:```<br>```#   99  – Missing input file```<br>```#```<br>```# User controllable options: None``` | *ex*-script DOCBLOCK |
| ```set –x``` | enable verbose logging |
| ```cpreq $COMIN/inputfile inputfile``` | copy essential input files into<br>working directory |
| ```export pgm=pmb_forecast``` | name of the binary executable |
| ```. prep_step```<br>```export FORT11=$FIXpmb/inputfile.tbl```<br>```export FORT12=inputfile```<br>```export FORT60=outputfile.grib2``` | clear FORTRAN unit assignments<br>set FORTRAN unit assignments |
| ```aprun –n 192 –N 24 $EXECmodel/$pgm >>$pgmout 2>errfile```<br>```export err=$?; err_chk``` | log program start<br>execute MPI program<br>error checking |
| ```if [ -s outputfile.grib2 ]; then```<br>```  cpfs outputfile.grib2 $COMOUT/outputfile.grib2``` | check for required output<br>copy output file to output |

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

### 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
  4. to install all executables:
        ./install_pmb.sh
  5. to clean sorc directory:
        ./clean_pmb.sh
```

### Example 11:  build script sorc/build_pmb.sh

sorc/install_pmb.sh and sorc/clean_pmb.sh are identical except replace "make" with "make install" and "make clean", respectively. These scripts can be combined into a single script using arguments.

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

| | |
|---|---|
| ```  done fi ``` | |

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

| | |
|---|---|
| ```  #%Module################################################ #                                  First.Last@noaa.gov #                                  ORGANIZATION # PMB-FCST v1.1.0 ######################################################## proc ModulesHelp { } {    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"   set    ver v1.1.0 setenv COMP intel setenv FC ftn  # Known 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 module load craype-haswell  # 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 ``` | module DOCBLOCK<br><br><br><br><br>module help<br><br><br><br><br><br>module description<br><br>set version and compiler variables<br><br>establish known conflicts<br><br><br><br><br>load ics and all ncep library modules used in the build process |

**Example 13: sorc/pmb_forecast.fd/makefile**

| | |
|---|---|
| ```  ############################################################ #  Makefile for xxx #  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 LDFLAGS = -O -convert big_endian BINDIR  = ../../exec ``` | Makefile DOCBLOCK containing instructions and use<br><br><br><br><br><br><br><br><br><br><br>name of compiler options of the loader 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   = -DDEBUG -g -traceback -check all                       debug options
FFLAGS  = -O3 -I $(INC)                                          compiler options

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

all: $(CMD)


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

debug:
        FFLAGS += $(DEBUG)

test:
        $(CMD) < input.txt > output.txt
        diff output.txt valid_output.txt

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

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

# Appendix B:  Variables and Directory Structure Tables

## Table 6: Binary executable production utilities accessible via module variables

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

**Table 7: Structure of *GESROOT* 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 sub-directories under com**

| Subdirectory | Description |
|---|---|
| $COMROOT/$NET/$envir/$RUN.$PDY/wmo | WMO headed output products |
| $COMROOT/$NET/$envir/$RUN.$PDY/gempak | gempak output products |

**Table 9: Structure of *DCOMROOT* directory on Phase 1**

| 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* | BUFR data tanks |

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

**Table 10: Structure of *DCOMROOT* directory on Dell**

| Subdirectory | Description |
|---|---|
| $DCOMROOT/$envir/YYYYMMDD | incoming data for one day |
| $DCOMROOT/$envir/YYYYMM | Incoming data for one month (select types only) |
| $DCOMROOT/$envir/*YYYYMMDD*/b*TTT*/xx*SSS* | BUFR data tanks |

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