NCEP Central Operations

WCOSS Implementation Standards

January 19, 2022

Version 11.0.0

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

I. Introduction

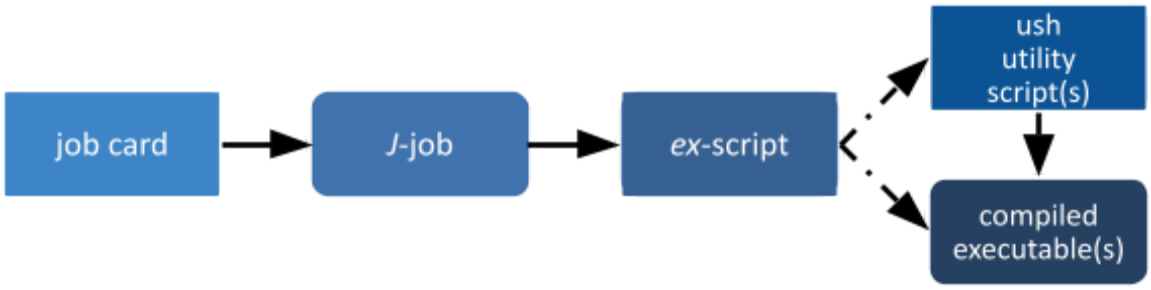
The reliable production and availability of the National Center for Environmental Prediction's (NCEP) guidance products plays a critical role in the mission of the National Weather Service to provide forecasts and warnings “for the protection of life and property and the enhancement of the national economy.” 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 Operations’ (NCO) Implementation and Data Services Branch (IDSB). WCOSS is currently composed of a GDIT managed Cray-EX cluster located in Manassas, VA and Phoenix, AZ. 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, PBS Pro, 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 *PBS* (job scheduler) directives and much of the execution environment (see Section III-A) and call the *J*-job to execute processing. The processing of the ecFlow script handles the substitution of ecFlow variables and files accessed via "%include" statements; the resulting job card is then handed off to PBS Pro via *qsub*.

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 the *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. 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
envir	Set to "test" during the initial testing phase, "para" when running in parallel (on a schedule), and "prod" in production.	job card
PACKAGEROOT	Root directory for the application, e.g. $OPSROOT/packages	job card
OPSROOT	Operations root directory, e.g. /lfs/$FS/ops/$envir	job card
job	Unique job name (unique per cycle)	job card
jobid	Unique job identifier, typically `$job.$PBS_JOBID`	job card
NET	Model name (first level of com directory structure)	*J*-job
RUN	Name of model run (third level of com directory structure)	*J*-job
PDY	Date in YYYYMMDD format	*J*-job
PDYm#	Dates of a previous day in YYYYMMDD format ($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, typically $OPSROOT/tmp in production	job card
DATA	Location of the job working directory, typically `$DATAROOT/$jobid`	*J*-job

HOME*model*	Application home directory, typically $PACKAGEROOT/*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
COMROOT	com root directory for input/output data on current system, typically $OPSROOT/com	job card
COMIN	com directory for current model's input data, typically $COMROOT/$NET/$*model*_ver/$RUN.$PDY	*J*-job
COMOUT	com directory for current model's output data, typically $COMROOT/$NET/$*model*_ver/$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
DCOMROOT	dcom root directory, typically $OPSROOT/dcom	job card
DCOMIN	dcom directory for current model's input data	*J*-job
DCOMIN*datatype*	dcom directory for incoming data from datatype *datatype*	*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 WCOSS2	job card
SENDDBN_NTC	Boolean† variable used to control sending products with WMO headers off WCOSS2	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**	version number of package in three digits; where *package* is the model's directory name	job card
***module*_ver**	Version of module *module* which is used at runtime by model *model*	version file
***extmodel*_ver**	version of external model dependencies; specified with two digit version number	version file
KEEPDATA	Boolean† variable used to specify whether or not the working directory should be kept upon successful job completion.	job card
MAILTO	List of email addresses to send email to	job card
MAILCC	List of email addresses to cc on email	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 adhere to the following:

For all model types:

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. Use a leading 0 in describing a grid resolution that is less than 1.
4. Include an “f” in front of the forecast hour
5. Pad forecast hours with zeros so that all files have the same number of digits
6. In cases where there is no forecast hour, but rather it is output that is before the cycle time, “tm” may be substituted for “f” in the filename.
7. *domain* does not need to be included in the filename if there is only one domain used by the model
8. ASCII inventory files (output of wgrib/wgrib2) should end with the extension “.grib2.idx”. Ex. hrrr.t10z.wrfnatf01.grib2.idx
9. Other index files (in binary format) should end with the extension “.bin.idx”
10. If *var_info* includes multiple pieces of information, they should be separated with a period. This includes resolution if there are multiple resolutions produced. Ex. gefs.t06z.avg.pres_a.0p50.f006.grib2, etss.t00z.stormsurge.2p5km.conus.grib2
11. 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.
12. Public products can be produced in any of the following formats: netcdf, bufr, grib2, ascii
13. If the directory structure includes the cycle, it should be a subdirectory. <model>.YYYYMMDD/HH/

For coupled models in particular:

14. Output directory structure should have subdirectories for each model component Ex. gefs.YYYYMMDD/HH/atmos/

Filename format for files in **com**:

non-ensemble: atmospheric, hydro models: *model.tHHz.var_info.f###.domain.format*

ensemble: atmospheric, hydro models: *model.tHHz.ens_mem.var_info.f###.domain.format*

non-ensemble: coupled models: *model.component.tHHz.var_info.f###.domain.format*

ensemble: coupled models: *model.component.tHHz.ens_mem.var_info.f###.domain.format*

hurricane models: *model.tHHz.storm_name.var_info.f###.domain.format*

space weather models: *model.var_info.valid_time.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 (227 is the domain in this case)

sref.t*HH*z.pgrb216.p10_3hrly.grib2.idx → sref.t*HH*z.p10.pres_3hrly.216.grib2.idx

gefs.chem.t*HH*z.a2d_0p25.f###.grib2 → gefs.chem.t*HH*z.a2d.0p25.f###.grib2

Filename format for files in the **wmo** sub-directory:

*format.model.*t*HH*z.*awp_var_info*.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

The utilities listed below must be used to assist in accomplishing certain tasks for all WCOSS models. They are accessible through the *prod_util* module. This module will put the below utility scripts in your environment's PATH and define other useful environment variables. The module is automatically loaded in all production jobs and should be loaded in development job cards. 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 `stdout`.

postmsg *

`postmsg` writes a message to a log file. The first argument is the log file name and the second is the message. The log file will default to stdout.

*startmsg and postmsg are no longer required in operations but the utilities will continue to be maintained.

err_chk/err_exit

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; jobs should fail with `err_chk` or `err_exit` as soon as a fatal error is encountered.

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, err_chk does nothing and job execution continues. If $err is non-zero, the job is aborted.

err_exit will write an error message with the time of the error, and immediately abort the job in PBS Pro. It accepts an error string as input to which it will prepend "FATAL ERROR."

cpreq

cpreq is used to copy files that are essential to an application. If the copy is unsuccessful for any reason, then a FATAL ERROR will be printed and the job will abort immediately. It has the same usage as the standard cp command.

cpfs

cpfs is used to copy files while ensuring that the whole file has been copied before it becomes accessible so that downstream applications will not attempt to copy or read a partial file. It has 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 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
```

cpfs calls the err_exit utility if either the cp or mv step returns non-zero status. However, as a further check, verify that a 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.

compath.py

The compath.py utility is used to discover the current absolute path of a given **com** directory and is used to set COMIN and COMOUT variables in *J*-jobs. 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 $envir/$NET/$model_ver/$RUN.$PDY)}
COMINm1=${COMINm1:-$(compath.py $envir/$NET/$model_ver/$RUN.$PDYm1)}
COMINgfs=${COMINgfs:-$(compath.py $envir/gfs/$gfs_ver/gfs.$PDY)}
```

COMOUT=${COMOUT:-$(compath.py -o $NET/$*model*_ver/$RUN.$PDY)}Run 'compath.py --help' to see all usage options. 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/prod/com/gfs"
```

mail.py

When nonfatal errors occur that may impact the quality of the model output, such as when backup data is used, it is important to notify the appropriate parties so that the error can be addressed. The `mail.py` utility is used to send 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
```

An addressee list can be included on the command line or set in advance via environment variable $MAILTO. To copy someone, use the "-c" flag:

```
echo "$msg" | mail.py -c <someones_email_address>
```

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

`getsystem` simply tells you which WCOSS system you are on. This utility exists for command line execution and must not be used in any operational packages. 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 output

System	*no flags*	–p
Dogwood phase 1	Dogwood	Dogwood-p1
Cactus phase 1	Cactus	Cactus-p1

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`. **Example 1: Using finddate.sh**

Script

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

PDY=20220101

# 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 20200101
The date ten days ago was 20211222
The date in ten days will be 20220111
The last four days were 20211231 20211230 20211229 20211228
The next four days are 20220102 20220103 20220104 20220105
```

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/$prod_util_ver


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 2022010112
 The date ten days ago was 2021122212
 The date in ten days will be 2022011112
 Six hours from now will be 2022010118
 Six hours ago was 2022010106
```

setpdy.sh

`setpdy.sh` creates a file PDY that is sourced to export the standard date variables PDYmn_m, PDYm{n_m-1}, PDYm{n_m-2}, …, PDYm2, PDYm1, PDY, PDYp1, PDYp2, …, PDYp{n_p-2}, PDYp{n_p-1}, PDYpn_p. By default, n_m and n_p are 7 but can be altered by providing alternate numbers as input parameters. The variable `cycle` must be set (in 't*HH*z' 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/$prod_util_ver
 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* and *wgrib2* modules. 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/$grib_util_ver
module load wgrib2/$wgrib2_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 and recovery. 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. All scripts that depend on the existence of a certain type of input or restart data to successfully run must check for the existence of such data before running and report an informative fatal error if the needed data is missing.

v. Recovery from code failure or abnormal system failure

Restart capability must be applied to an operational job 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 part with the problem needs to be rerun, thus the time to completion is shorter. An example of this would be to submit a separate post-processing job for each forecast hour, so any failure for one forecast hour does not impact others, and can be recovered from quickly. 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. For a forecast job, this would involve writing out checkpoint or restart files at fixed intervals during the forecast, from which the model can be restarted.

The job scripts must be designed so this restart will happen automatically if the job is rerun. Any products delivered by a restarted production job must not be delayed by more than 15 minutes. Data assimilation jobs are exempt from this requirement, but steps should be taken to minimize runtimes and enhance re-runnability of these processes.

vi. No background processing

PBS Pro loses control of processes when they are put in the background. Therefore, background processing must be avoided. Killing a PBS Pro job must terminate all processes running under it.

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`. 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. Text formatting

All text files (scripts, source code, config files, etc.), as well as standard output for all jobs/scripts, must only use the basic ASCII character set, with no Windows-format carriage returns, stylized quotation marks, or other non-standard characters.

x. 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).

xi. Points of contact

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

xii. Cold starts

All jobs that depend on restart data from previous runs must include a cold restart option. Cold start is the ability to run using the current inputs and observations without any data from previous runs. The cold start option must be activated by the addition of "export COLDSTART=YES" in the job card

xiii. Removal of dead code

After initial coding updates/debugging efforts, executable statements that are made inert by commenting must be removed. Rely on configuration management software for content differentials.

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 latest available version of the Intel or Cray (`cc`, CC, and `ftn`) compiler collections.
3. All libraries must be approved for production use. Approved libraries are found by running "`module avail`" in a default environment. Hidden modules are not allowed to be used in production. Makefiles must only include compilers and libraries using variables defined in modules:

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

   ```
   module load cpe-cray
   module load intel/$intel_ver
   ```

```
module load w3nco/$w3nco_ver
```

Within the makefile:

```
LIBS = ${W3NCO_LIB4}
ndate: ndate.f
       $FC -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 source code or build system. If a module file does not provide a certain desired variable, the necessary value should be derived from the module file's contents programmatically as opposed to hardcoded (e.g., when using `bufr` module, use "$BUFR_INC4/bufrlib.h" not "/lfs/h1/ops/prod/libs/intel/19.1.1.217/bufr/11.4.0/include_d/bufrlib.h"). This way, if a module version is upgraded, no further modifications will be necessary for the code to compile and run with the appropriate libraries and executables.
5. Code must compile without errors or warnings. Errors and warnings may not be suppressed, and the compiler warning level ("-W" options) must be *at least* the default one.
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) Paths of files outside a job's working directory (e.g., input data from COMIN or DCOM) 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. For example, structured control must be used instead of GO TO statements, and code must be well documented.
10. Executables should be built with production compilation settings and tested for and ridded of memory leaks/allocation problems with, e.g., valgrind4hpc

C. Interpreted Code (bash, ksh, perl, or python 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** shell script:

   ```
   set -x
   ```

   and add timing info to the execution trace by including the following in the *J-job*:

   ```
   export PS4='+ $SECONDS + '
   ```
2. setpdy.sh must be called after cd to the working directory ($DATA)
3. Utilize standard environment variables and utilities (See Section III).
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`, if applicable, and `err_chk`.
6. Any executions that print verbose output (more than 100 lines or so per execution) must redirect standard output and standard error to a file under $DATA, for example:

   ```
   $EXECmodel/$pgm >> $pgmout 2> errfile
   ```
7. Production utilizes a centralized cleanup of directories in COMROOT. Production scripts must not remove directories at the $COMROOT/$NET/$ver/$RUN.$PDY level. Output must conform to the output structure of $COMROOT/$NET/$ver/$RUN.$PDY.

8. Do not assume that the current directory (“.”) will be in the execution path ($PATH). *(Invoke temporary script as $DATA/scriptx or ./scriptx).*
9. Model scripts and executables should be called explicitly, eg, $USHmodel/scriptx. ($USHmodel and $EXECmodel should not be added to $PATH).
10. Remove all references to developer work areas and all development tools (benchmarking, etc.) before submitting to IDSB.
11. If your application should continue if a preceding step fails, it must be documented in a comment in the script just before (or after) the relevant part is called and a descriptive “WARNING:” message printed to stdout.
12. Never write to **dcom**! Unless you run data ingest from an outside source.
13. Ensure that files containing restricted data are assigned the appropriate group and permissions.
14. There must be no false/misleading errors and no syntax errors in the standard output/error file.
15. Ensure all non-zero stops, aborts, calls to err_exit, etc are for good reason. (Eg, consider whether a bad observation should be skipped rather than causing the job to fail).
16. The interpreter must be added to the top of all shell scripts with a "#!" statement.
17. Shell scripts must be invoked directly *(eg, “<path_to_script>”, not “sh <path_to_script>“)*.
18. All packages that use Python scripts must specify a Python version through the module system, and must only call a Python executable that is from a module, not the system version. "module load python/${python_ver:?}" or similar must be present in all job files that will lead to python script calls, where the python version is defined in the version file. Python version must be at version 3 or higher.

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. Any alert that is new or changing needs to be coordinated with the Dataflow Team so that the product will continue to go to all of the external customers specified in the governing tables. 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, git 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 build script, and optionally a module file, used to set up the build environment. It must define the compiler and its version (by loading the appropriate versioned compiler), 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. The modulefile or script must not reference unused software.
5. WCOSS uses the Lmod environmental module system, therefore all module files must be in Lmod/Lua format
6. 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. The makefile must not include references to unused libraries. Example 13 of Appendix A contains an example. See Environment Equivalence (EE) standards for more details about builds
7. The resulting executable(s) must continue to work if the original build path is removed or renamed (eg, when moving the package from ops/para to ops/prod).
8. There are four critical targets that must be defined in every makefile. They are all, debug, install, and clean. Additionally, a test target is required to run unit tests for libraries and utility programs. Example 13 of Appendix A contains an example of each.

   a. The debug target must minimally contain the check all and ftrapuv flags in fortran or their equivalent in other accepted languages

9. 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*.v*X*.*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 that can be compiled
exec	binary executables
parm	parameter files
parm/wmo	Specific subdirectory under ./parm for WMO GRIB headers
fix	fixed fields, tables or other static input data
lib	model-specific libraries
ecf	ecFlow scripts and definition files
gempak	all gempak-related files
versions	contains run.ver and build.ver, which are files that get automatically sourced in order to track package versions at run time and compile time, respectively (e.g., "export bufr_ver=11.4.0; export gempak_ver=7.3.3"; export lmp_ver=v2.4.0. The “v” is excluded for module versions).
modulefiles/	model module files

Table 4 lists the primary data and application directories used within the WCOSS NCO production environment. These 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
prod	applications/packages in the production suite
test	applications/packages in the test suite (unscheduled)
para	applications/packages in the parallel suite (scheduled)
${envir}/com	data and application output, including outgoing products
${envir}/dcom	incoming data (retrieved from outside WCOSS)
${envir}/brs	backup of production packages
${envir}/tmp	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 PBS Pro job stdout and stderr. World Meteorological Organization (WMO) headed output products are placed in a model's com structure under a **wmo** subdirectory. Model output products in GEMPAK format (grids, model vertical profiles) are placed in the model's com structure under a **gempak** subdirectory. Table 5 (below), Table 7, Table 8, and Table 9 (in Appendix B) show the structures of com, and dcom directories, respectively.

Table 5: Structure of COM directories

/lfs/h1/ops/ subdirectory	Description
prod/com/NET/$ver/*RUN.YYYYMMDD*	production model output for a day
test/com/NET/$ver/*RUN.YYYYMMDD*	test model output for a day
para/com/NET/$ver/*RUN.YYYYMMDD*	parallel model output for a day
prod/com/output/*YYYYMMDD*	production job stdout/stderr for a day
test/com/output/*YYYYMMDD*	test job stdout/stderr for a day
para/com/output/*YYYYMMDD*	parallel job stdout/stderr for a day
prod/com/output/transfer/YYYYMMDD	transfer job stdout/stderr for a day
prod/com/output/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.

VII. 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 run.ver/build.ver

The version file tracks the versions of all packages and modules used by your application. It must not reference packages or modules that are not used.

`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 PBS Pro. 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).

`#PBS -N %E%pmb_forecast_00`	job name
`#PBS -A %PROJ%-%PROJENVIR%`	project identifier
`#PBS -q %QUEUE%`	PBS Pro queue name
`#PBS -S /bin/sh`	login shell
`#PBS -l walltime=01:00:00`	wall clock
`#PBS -l select=8`	Request 8 nodes
`export model=pmb` `%include <head.h>` `%include <envir-p1.h>`	begin ecFlow communication set up environment
`export cyc=%CYC%`	set the cycle
`export MPICH_GNI_MAX_EAGER_MSG_SIZE=65536` `export FORT_BUFFERED=TRUE`	define parallel environment variables
`module load util_shared/$util_shared_ver` `module load grib_util/$grib_util_ver`	load only modules need for this job
`$HOMEpmb/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
- SENDDBN
- SENDDBN_NTC
- KEEPDATA
- DBNROOT
- COREROOT
- SENDECF
- SENDCOM

Example 7: prod_envir module

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

```
$ module display prod_envir
-------------------------------------------------------------------
   /apps/ops/prod/nco/modulefiles/prod_envir/2.0.3.lua:
-------------------------------------------------------------------
setenv("OPSROOT","/lfs/h1/ops/prod")
setenv("OPSROOTssd","/lfs/f1/ops/prod")
setenv("COMROOT","/lfs/h1/ops/prod/com")
setenv("DATAROOT","/lfs/f1/ops/prod/tmp")
setenv("DCOMROOT","/lfs/h1/ops/prod/dcom")
setenv("PACKAGEROOT","/lfs/h1/ops/prod/packages")
```

Example 8: *J*-job JPMB_FORECAST

`#!/bin/sh`	
`date` `export PS4='+ $SECONDS + '` `set -x`	print starting time prepend time to output enable verbose logging
`export DATA=${DATA:-${DATAROOT:?}/${jobid:?}}` `mkdir -p $DATA` `cd $DATA`	create temporary working directory
`export cycle=${cycle:-t${cyc}z}` `setpdy.sh` `. ./PDY`	set up temporal variables, including PDY
`export SENDDBN=${SENDDBN:-YES}` `export SENDDBN_NTC=${SENDDBN_NTC:-YES}` `export SENDECF=${SENDECF:-YES}`	alert output via DBNet alert wmo output send signals to ecFlow
`export USHpmb=$HOMEpmb/ush` `export EXECpmb=$HOMEpmb/exec` `export PARMpmb=$HOMEpmb/parm` `export FIXpmb=$HOMEpmb/fix`	sub-directories of the current model
`export NET=${NET:-nco}` `export RUN=${RUN:-pmb}`	variables used in com directory organization
`export COMINgfs=${COMINgfs:-$(compath.py gfs/prod/gfs.$PDY)}` `export COMIN=${COMIN:-$(compath.py ${NET}/${envir}/$RUN.$PDY)`	locations of incoming data
`export COMOUT=${COMOUT:-$(compath.py -o ${NET}/${envir}/$RUN.$PDY)` `export COMOUTwmo=${COMOUTwmo:-${COMOUT}/wmo}` `export COMOUTgempak=${COMOUTwmo:-${COMOUT}/gempak}`	locations of outgoing data
`mkdir -p $COMOUT $COMOUTgempak $COMOUTwmo`	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

`if [ -e "$pgmout" ]; then` `  cat $pgmout` `fi`	print exec output
`if [ "${KEEPDATA^^}" != YES ]; then` `  rm –rf $DATA` `fi`	remove temporary working directory
`date`	print ending time

Example 9: *ex*-script expmb_forecast.sh

`#!/bin/sh`	
	ex-script DOCBLOCK
`# 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` `#`	enable verbose logging
`# User controllable options: None`	copy essential input files into working directory name of the binary executable
`set -x`	clear FORTRAN unit assignments set FORTRAN unit assignments
`cpreq $COMIN/inputfile inputfile`	log program start
`export pgm=pmb_forecast`	execute MPI program error checking
`. prep_step` `export FORT11=$FIXpmb/inputfile.tbl` `export FORT12=inputfile` `export FORT60=outputfile.grib2`	check for required output copy output file to output directory alert output file
`mpiexec <options> $EXECmodel/$pgm >>$pgmout 2>errfile` `export err=$?; err_chk`	

`<pbs_release_nodes -a>`	If multiple nodes were requested and the remainder of the job is serial processing, release the extra nodes to make them available to other jobs. See pbs_release_nodes man page for more options.
`if [ -s outputfile.grib2 ]; then` `  cpfs outputfile.grib2 $COMOUT/outputfile.grib2` `  if [ "${SENDDBN^^}" = YES ]; then` `  $DBNROOT/bin/dbn_alert MODEL PMB_FCST \` `  $job $COMOUT/outputfile.grib2` `  fi` `else` `  err_exit "outputfile.grib2 was not generated"` `fi`	terminate the job if the expected output cannot be found
`. prep_step` `export FORT11=outputfile.grib2` `export FORT51=grib2.t${cyc}.z.pmb.f000`	setup for tocgrib2 exec define input file define output file
`$TOCGRIB2 <$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`	add WMO header to file error checking

Example 10: build readme file sorc/README

```
Build instructions:
  1. cd to the sorc directory
  2. 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
  3. to install all executables:
       ./install_pmb.sh
  4. 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`	enable verbose logging
`module reset` `module use ../modulefiles` `module load build_pmb.module`	move to the source directory of the given executable make the executable
	print error message if build is unsuccessful

```
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
  done
fi
```

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

Example 12: modulefiles/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


# Load Cray parallel environment
module load cray-mpich/$::env(cray_mpich_ver)

# Load Intel programming environment
module load intel/$::env(intel_ver)
# Load NCEP libs modules
module load hdf5/$::env(hdf5_ver)
module load netcdf/$::env(netcdf_ver)
module load bacio/$::env(bacio_ver)
module load w3nco/$::env(w3nco_ver)
module load jasper/$::env(jasper_ver)
module load libpng/$::env(libpng_ver)
module load zlib/$::env(zlib_ver)
```

module DOCBLOCK

module help

module description

set version and
compiler variables

load intel and all ncep
library modules used
in the build process

Versions come from
sourcing
versions/build.ver
prior to loading
module

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 LDFLAGS = -O -convert big_endian BINDIR = ../../exec INC = ${G2_INC4} LIBS = ${G2_LIB4} ${W3NCO_LIB4} ${BACIO_LIB4} ${JASPER_LIB} ${PNG_LIB} ${Z_LIB} CMD = pmb_forecast DEBUG = -check all -ftrapuv FFLAGS = -g -traceback -O3 -I $(INC) # 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) ```	Makefile DOCBLOCK containing instructions and use name of compiler options of the loader executable location include files libraries (variables from Lmod modules) executable name debug options compiler options

VIII. 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	wgrib2
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 8: Structure of sub-directories under com

Subdirectory	Description
$COMROOT/$NET/$model_ver/$RUN.$PDY/wmo	WMO headed output products
$COMROOT/$NET/$model_ver/$RUN.$PDY/gempak	gempak output products

Table 9: Structure of *DCOMROOT* directory

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

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