March 6, 2013

MEMORANDUM FOR: WCOSS Programmers
               FROM: NCEP/Central Operations/Production Management Branch
               SUBJECT: Technical policy for implementation standards on the Central
                               Computer System (CCS)

This document outlines policies and technical guidelines for consideration when implementing operational computer code or "numerical models" on the WCOSS. The formal process to request a change or addition is a Request For Change (RFC) and is discussed in another document. However, during the transition from CCS to WCOSS, developers should use the WCOSS Transition Request form, v 1.0.0, which is available from the Centers' Transition Points of Contact.

The goal of this document is to discuss and provide examples of operational quality code and scripts, set forth best practices in coding, establish a common base of coding standards, and provide a means for improving coding techniques and best practices.

Most often, a request for change will be composed of changes to compiled code and changes to scripts and other interpreted files that call the compiled executable code. It is recommended that the scripts and executable code called by those scripts be grouped together in some logical fashion. Many developers will replicate the directory structure of /nwprod when organizing the files, scripts, and executable source that comprise the requested change. While this is not mandatory, and may be governed by the complexity of the change requested, it will be easier for the production staff to implement if the location of the changes is easily understood and familiar.

Code Delivery guidelines - Section 1 (Source code / compilable code / binary executable code)
a) All of the files needed to completely build the executable should be available in one place on the WCOSS. This does not include system or standard production libraries, but the make process should contain references to standard production libraries found in /nwprod/lib and standard system libraries.

b) Use a readme file in the top level source directory to explain the build process if it requires choices be made during the compile or if it is in any way non-standard. An explanation of how to build in the same directory as the source will eliminate confusion and errors if it becomes necessary to rebuild the executable to resolve a production failure or other emergency situation.

Further discussion

1. Realizing that there are some constraints on how source code can be packaged, it is advantageous for the production implementation team to be able to understand how the pieces fit together. Source code that forms an executable or group of related executables should be contained in a directory; and while that directory may contain other sub-directories, the compilation scripts, makefiles, and documentation for building the executable should be easily understandable. If there are specific sequences of scripts to be run and options that need to be chosen during the build, they should be clearly specified in a readme file in the top level source directory.

2. The source code directory must be available on the WCOSS and must be accessible by NCO's implementation staff. (Do not put files in HPSS). All of the source code (main program and subroutines) required to execute your program must be included in this directory as separate files (no concatenated files). We will not selectively copy routines from your directory. We will not include or link source or executables from private libraries. To ensure that the latest version of the source code is used, programmers should start with, and modify, the operational source code which they have copied from /nwprod/sorc.

3. It is preferable for the top level source code directory to have a makefile that does everything needed to build the executable. It is also preferable to have the executable name be the same as the source directory that will contain it; in this way automated scripts can batch process the building of executables and move them to the main executable directory when a mass build is needed.

4. When the make process produces more than one executable please make sure that the final step copies the executables to the top level source directory and their filenames are correct for all the scripts that use them. If different compiler options are needed for each subroutine, then it is the programmer's responsibility to include these options in the makefile.

5. C and Fortran compilers should be Intel (ifort and icc). If you must use the gnu compilers, be prepared to make your case as to why these are needed over the Intel compilers.

Documentation Blocks (DOCBLOCKS)
The goal of documentation should be to help understand what the code does. From a production perspective, documentation blocks can help troubleshoot a problem and help the staff remedy a problem more quickly. Sometimes too much information is as bad as none at all. Below is a suggested format and information outline for a docblock. We ask that you use your judgment in what information will be of most help and include it within your code.

```
#### UNIX Script Documentation Block
#
# Script Name:
# RFC Contact:
# Abstract:
#
# Script History Log:
#
# Usage: <Specify typical arguments passed>
#   Script Parameters:
#   Modules and Files referenced:
#     scripts: <file names of scripts called by this script>
#     parms: <file names in the parm directory the script uses>
#     fix:      <file names in the fix directory the script uses>
#     executables:  <compiled code this script calls>
#
# Condition codes:
#     < list any exit condition or error codes the script returns if
```

```
#      any >
#     If appropriate, descriptive troubleshooting instructions or
#     likely causes for failures could be mentioned here with the
#     appropriate error code
#
# User controllable options: <if applicable>
#
# Attributes:
#     Language:  RedHat Linux
#     Machine:    NCEP WCOSS
#
```

Makefiles

Makefiles provide the rules to the "make" command, which creates the executable from source code(s).

Makefiles should create one executable. The name of the executable and the name of the directory containing the source code to make the executable should be named the same with the addition of the appropriate extension (.fd, .cd denoting Fortran or C code respectively) at the end of the directory name.

Wx_control2rule_the_world.fd would therefore contain Fortran code and a makefile to produce the wx_control2rule_the_world executable which would be created in the wx_control2rule_the_world.fd directory when compiled.

Complex makefiles that require configuration steps should have a README file detailing those step-by-step instructions.

Makefiles that must produce more than one executable should also have a README file explaining this, and the executables produced should have the correct name as specified in the RFC and README. The executables should be available in the top level source directory when the compilation is complete.

If a makefile has a dependency on another code (for example, the NAM post-processor code uses WRF-specific libraries created when the NAM forecast model is compiled), that code must either already exist in production or be part of the RFC package. Detailed instructions should be provided to NCO, instructing them as to what code in the RFC package should be compiled first if such dependencies exist.

Please do not specify an absolute path outside of the source code directory to copy executables, libraries, or any other products. NCO will compile and test program functionality in parallel and test environments before implementing into production.

The following makefile shows the preferred format, but programmers are free to create (and test) their own.

Makefile example, where TARGET=the name of your code:

```
################################################################
########
#       Makefile for xxx <This is the Documentation Block containing
instructions and use>
#       Use:
#       make            -  build the executable
#       make clean   -  start with a clean slate
################################################################
########
# Define the name of the executable
TARGET = your_model # The name of the executable produced #
#
# CPP, Compiler, and Linker Options
FC        = ifort # Fortran compiler used #
CPP       = icpc
CPPFLAGS =
# Optional compiler directives
OPTS      = -auto -xHost
LIST      =
FREE      =
FIXED     = -fixed
TRAPS     =
PROFILE   =
DEBUG     =
MEM       =
MAP       =
W3LIBDIR = /nwprod/lib

# There may be other definitions listed here
# depending on the needs of the code to produce the executable
#
# Assemble Options
FFLAGS   = $(OPTS) $(LIST) $(TRAPS) $(PROFILE) $(DEBUG)
FFLAGST  = $(OPTS) $(LIST) $(FREE) $(TRAPS) $(PROFILE) $(DEBUG)
LDFLAGS  = $(MEM) $(MAP) $(SMP) $(PROFILE)
LIBS     = $(SEARCH) $(NCDLIBS) -L$(W3LIBDIR) -lw3nco_4 -lw3emc_4
-lbacio_4
# Threaded object files
OBJST=  WRFBDYGEN.o
#
# Non-threaded object files
OBJS=   COMBC.o
```

```
#
# Includes
INCLUDES= parmeta.res
#
# Common Blocks
COMMS=
DEPS= $(COMMS) $(INCLUDES)
.SUFFIXES:        .F .f .o
.F.f:
        $(CPP) $(CPPFLAGS) $< > $*.f

$(TARGET):        $(OBJS) $(OBJST)
        $(FC) $(LDFLAGS) -o $@ $(OBJS) $(OBJST) $(LIBS)

$(OBJS):         $(DEPS)
        $(FC) $(FFLAGS) -c $<

$(OBJST):        $(DEPS)
        $(FC) $(FFLAGST) -c $<

clean:
        /bin/rm -f  $(TARGET) *.lst *.o *.mod
# End of sample makefile
```

This is only intended as an example. Different formats are acceptable as long as they are easily understood and correctly produce the intended executable.

Fortran Unit Number Assignments
We understand that some application source code is used by a community of scientists and numerical modelers, and that it is impractical to assign specific unit numbers to files used in Fortran executables. However, in code that still uses static units, and where the flow of operation is simple, please make an effort to use a standard or consistent assignment strategy.
It is useful to have a consistent standard for all input and output across all programs to aid in troubleshooting failures and provide a means to quickly understand how data is being used.
As an example, the following convention may be helpful:
1) Units 1 through 4, 7 through 10, and 50 are reserved for future use.
2) Use units 5, 11-49 for all INPUT files; i.e., all files containing data created prior to the execution of the program.
3) Use units 6, 51-79 for all OUTPUT files; i.e., all files containing data for subsequent programs to use.
4) Use units 80-94 for all WORK files; i.e., all files that are written and read in the same program but have no further use.
Except for work files, the same unit number should NEVER be used for both input and output by the same program.

Note:

Users should associate filenames to unit numbers in the shell script prior to program execution. On the IBM WCOSS, users should us the environmental variable FORT*k*, where *k* is a 2 digit number (if unit number is less than 10, use a leading 0). Note that on the CCS, under XL Fortran's runtime environment, setting unit_vars=yes let one connect a file of one's choosing to Fortran unit *k* (opened without a filename, of course) by assigning its name to the environment variable XLFUNIT_*k*. Under Intel Fortran, one can achieve the same end by using environment variable FORT*k*. No other environment variable needs to be set for the Intel runtime library for this to work. Filenames should not be hardwired in the source code.

Example:
    export FORT16="inputfilename"
    export FORT60="outputfilename"

Code Delivery guidelines - Section 2 (Scripts and other interpreted files)

1. The location of the script necessary to execute the code must also be listed on the RFC if the script is being changed. To ensure that the latest version of the script is used, programmers should start with the operational scripts which they have copied from /nwprod and then make modifications to those scripts in their development workspace.

2. Your directory must also contain any new or modified parameters or fixed fields needed by the code.

3. When preparing operational scripts, the following standard must be followed. See Attachment III for examples and general information which will be helpful in developing production scripts.

   a. Use Bash (/bin/sh, which defaults to /bin/bash on WCOSS) or Korn Shell (/bin/ksh) .
   b. Obtain the NCEP production dates by using the setpdy.sh production utility located in /nwprod/util/ush.
   c. Logging must be turned on via a "set -x" command at the top of the script.
   d. Utilize standard environment variables (See Table 1-5 in Attachment III).
   e. Utilize standard production file and naming conventions.
   f. Each block of copies from the scratch directory to /com,/nwges or /pcom must be wrapped with logic testing for the presence of the variable SENDCOM.
   g. Each block of dbnet alerts must be wrapped with logic testing for the presence of the variable SENDDBN.
   h. Each execution of a C or Fortran code must be wrapped with the use of production utilities prep_step, startmsg and err_chk. The standard error should be redirected to a file named errfile in the current working directory. The standard output of each execution should be appended to $pgmout (standard production variable).
   i. In each system of scripts, the top level script creates and initializes the standard production environmental variables and resides in /nwprod/jobs with the standard naming convention of JXXXX. The top level J-job script calls the main driver script which resides in /nwprod/scripts with the standard naming convention of exXXX.sh.ecf. Any needed sub-scripts to the main driver script will be located in /nwprod/ush or /nwprod/util/ush.

  j. Production utilizes a centralized cleanup and creation of directories in /com and /nwges.
  Production scripts should not remove or create production directories at the
  /com/$NET/$envir/$RUN.$PDY level.
  k. Remove development tools such as hpmcount before submitting to PMB.
  l. Remove references to developer work areas.

Code should be written from an operational perspective

Diagnosing failures quickly is a necessary requirement of the operational staff. To that end, all code should be scrutinized for 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;

Notification of use of backup data

For scripts that have a secondary (or beyond) data source to be used when the primary data is not available, the script should include a message that indicates the primary data is not available and backup data is being used. If use of backup data is a concern to the developer (i.e. if continued use will result in a degraded product), he/she should include code in the script to email a message re: the use of the backup data.

Descriptive error messages

Executable code should be written so that if a failure occurs, the context of that failure is communicated as descriptively as possible. Failures should not be allowed to propagate downstream of the point where the problem can be detected.

Appropriate modes of failure

An executable should not terminate abnormally with a segmentation or memory fault for errors that are discoverable / trappable. For example, lack of input data should be handled either in the script before the executable runs, or by the executable if checking in the script is not practical. If a script is able to use a backup form of input data (in case the primary data is unavailable), the developer should design the script to use the backup data and supply a meaningful message indicating the use of the backup data. For cases where it is undesirable to use the backup data for more than a short period of time, the developer should work with NCO's SPA team to include coding in the script which checks for the presence of the primary data and alerts NCO and/or the developer if that data is unavailable.

Saving standard output

To ensure that all standard output from the code get saved (whether it fails or not) users should make sure that the standard output file (called $pgmout) is written at the end of the job, by adding "cat $pgmout" at the end of the top-level J-job script. If the standard output file is very large, users should reduce the number of print statements in their codes to make the standard output file size more manageable.

Error handling

Proposed set of error codes for at least some categories of errors - this may be as simple as a list of numerical codes resulting from calling STOP XXX in Fortran, or exit(XXX) in C where XXX is a numerical code agreed upon by EMC and NCO for certain classes of errors. This will enable the operations staff to more quickly and accurately determine the failure mode and hopefully resolve the problem.

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

Pre-delivery testing and SPA testing
All codes/scripts must be tested by the developer prior to submission of one or more Requests for Change (RFC) for production implementation. The PMB Senior Production Analyst (SPA) team lead will review submitted RFCs for completeness and will then assign the RFCs to a SPA. Incomplete RFCs may be returned to the developer. The SPA will then set up the code/scripts in either the nwtest or nwpara environment for pre-implementation testing. The SPA will contact the developer with any questions and will ask the developer to validate the output from any tests run prior to implementation.

Code Delivery Guidelines Section 3 - (Production directory structure and Utilities)

Production Directory Structure
Table 1-1 shows an overview of the directory structure necessary to run production.

Table 1-1 Directory Structure

Directory	Description
/nwprod	Production Applications
/nwtest	Test Applications
/nwpara	Parallel Applications
/nwbkup	Backup of Production Applications
/nwges	Model Spin-up Data
/com	Data and Application Output, Including Outgoing Products
/dcom	Incoming Data
/pcom	Outgoing Products with WMO Headers

Structure of Application Directories
Table 1-2 shows an overview of the application directories. The directory names are the sub-directories within /nwprod, /nwtest and /nwpara.

Table 1-2 Application directories

Directory	Description
Jobs	Wrapper Scripts (J-Jobs)
Scripts	Main Driver Scripts (ex-scripts)
Ush	Utility Scripts
Fix	Static Input Data
Parm	Static Input Data

Exec	Executables
Sorc	Source Code
Util	Utilities spanning multiple applications

Structure of /nwges Directory

Several of the weather forecast models running in production produce output to be used later as input for subsequent model runs. This select set of critical output data used to begin model runs is often referred to as model guess fields. The model guess fields are stored in /nwges. Table 1-3 shows the directory structure of /nwges.

Table 1-3 /nwges Directory Structure

Directory	Description
prod/*model_name.YYYMMDD*	Production Spin-up data for model
test/*model_name.YYYMMDD*	Test Spin-up data for model
para/*model_name.YYYMMDD*	Parallel Spin-up data for model

Structure of /com Directory

The /com directory contains output data, stdout and stderr from production jobs. The default resident time for data in /com is ten days. Table 1-4 shows the directory structure of /com.

Table 1-4 /com Directory Structure

Directory	Description
model_name/prod/net_nam.YYYYMMDD	Production Model Output for a day
model_name/test/net_name.YYYYMMDD	Test Model Output for a day
model_name/para/net_nam.YYYYMMDD	Parallel Model Output for a day
output/prod/YYYYMMDD	Production job stdout/stderr for a day
output/test/YYYYMMDD	Test job stdout/stderr for a day
output/para/YYYYMMDD	Parallel job stdout/stderr for a day
Logs	Log files

Standard Environmental Variables

Inside of the production scripts there are environmental variables reserved for production use. A majority of the production utilities rely on the use of these standard variables. These variables are set inside of the production wrapper scripts. Unsetting or programming around these variables inside of the driver and supporting scripts may result in an undesired job outcome making it difficult to troubleshoot. Table 1-5 shows the list of the standard environmental variables used in production.

Table 1-5 Standard Environmental Variables

Variable Name	Description
PDY	Today's Date formatted YYYYMMDD
PDYm1-7	Date 1-7 days ago formatted YYYYMMDD
PDYp1-7	Date 1-7 days ahead formatted YYYYMMDD
DATA	Temporary Working Directory
Jlogfile	Logfile of start time, end time, and error messages of all jobs
Outid	Job ID appearing in jlogfile

Jobid	Name of stdout file for all programs in a job
Cycle	Model Cycle time formatted tHHz
Cyc	Model Cycle time formatted HH
SENDCOM	Enable/Disable file copying to /com
SENDDBN	Enable/Disable DBNet Alerts
SENDECF	Enable/Disable ecFlow hooks
NET	Model Name
RUN	Type of model run
Pcom	Directory for copies to /pcom
COMIN	/com directory for data input
COMOUT	/com directory for data output
GESdir	/nwges directory for read and write
utilities	Directory containing utility scripts
utilexec	Directory containing utility executables
EXEC*model_name*	Directory containing model executables
FIX*model_name*	Directory containing model fix files
PARM*model_name*	Directory containing model parameter fields
USH*model_name*	Directory containing supporting model scripts
ECFBIN	Directory containing ecFlow executables
Envir	= Prod, para, or test. Set in production ECF script; do not reset in subscripts

Standard File Naming Conventions

Production file names should represent the name of the model run, the cycle of the model run, the type of data the file contains and the forecast hour the data represents. Filenames should not contain the date as the directory in which it resides already represents the date. Filenames should not contain uppercase characters.

Example:
    gfs.t${cyc}z.pgrbf${fhr}   where cyc is the cycle and fhr is the forecast hour.

Basic Production Utilities

There are several utilities available in production to help you incorporate the basic job functionality required to meet operational standards. This section is intended to introduce you to the basic utilities used by most production jobs.

Date Utilities

Developers should use the production date utilities and should ensure their scripts will work as expected when crossing over the new year. The following examples show how to use the production date utilities.

finddate.sh

Given a date, *finddate.sh* will return date a specified number of days before or after the provided date. *finddate.sh* will also provide a sequence of dates leading to the specified number of days before or after the provided date. Example 2-1 shows how to use *finddate.sh*.

Example 2-1 Script Using finddate.sh

```
#!/bin/sh
utilscript=/nwprod/util/ush

today=20120101

# Single Date Example
ten_days_ago=`sh $utilscript/finddate.sh $today d-10`
ten_days_ahead=`sh $utilscript/finddate.sh $today d+10`

# Sequence Example
last_four_days=`sh $utilscript/finddate.sh $today s-4`
next_four_days=`sh $utilscript/finddate.sh $today s+4`

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

Example 2-1 Output
Today's Date is 20120101

The date ten days ago was 20111222
The date in ten days will be 20120111

The last four days were 20111231 20111230 20111229 20111228
The next four days are 20120102 20120103 20120104 20120105

setpdy.sh
This script helps you set the variables PDYm1-7, PDY and PDYp1-7. This utility will output a file PDY
in the current working directory which can be sourced in the parent script to set the PDY variables.
setpdy.sh expects the environmental variable **cycle** to be set when executed. The default centered date is
the current day's date. If the environmental variable PDY is set when executed, the centered date will be
the value of PDY.

Script *setpdy.sh* uses date files in /com/date set by ecFlow production jobs /prod00/ncepibm00/j100_00
and /prod12/ncepibm12/j100_12, run at 2330 UTC and 1130 UTC respectively. At 2330 UTC, the date
files for cycles 00-11 UTC are set ahead to the next day. At 1130 UTC the date files for cycles 12-23
UTC are set ahead to the next day. Therefore, if you were to set cycle to t12z and run setpdy.sh between
2230 and 1130 UTC, you would get a PDY file centered on the previous day's date. This is because the
12 UTC cycle has not started. This has been done by design to allow 12 UTC production jobs to be run
late into the 00 UTC cycle. Example 2-2 shows how to use *setpdy.sh*.

Example 2-2 Script Using setpdy.sh

```
#!/bin/sh

#note that on WCOSS, /bin/sh defaults to bash.

export utilscript=/nwprod/util/ush

# If PDY is not set, the dates would be centered based off the current cycle date.
# Try running with PDY not set to see what happens.
export PDY=20120101
export cycle=t12z
$utilscript/setpdy.sh
. PDY
```

Example 2-2 Contents of File PDY
```
export PDYm7=20111225
export PDYm6=20111226
export PDYm5=20111227
export PDYm4=20111228
export PDYm3=20111229
export PDYm2=20111230
export PDYm1=20111231
export PDY=20120101
export PDYp1=20120102
export PDYp2=20120103
export PDYp3=20120104
export PDYp4=20120105
export PDYp5=20120106
export PDYp6=20120107
export PDYp7=20120108
```

Logging and Error Checking Utilities
All production scripts must adhere to a standard error checking methodology. The reasoning behind this is to avoid lost time in having to re-run preceding jobs when failures occur. The earlier a failure can be caught, the less time it takes to recover from that failure.

Providing notification that a part of a sequence of job steps has failed should be a logical process. If a subsequent job or part of your application depends on the successful completion of a prior executable or processing operation, then that dependency must be checked for successful completion and a failure message returned if it does not. There are no exceptions to this rule.

If your application can continue if a preceding step fails, it should be documented in a comment in the script just before or after the relevant part is called.

setup.sh
To properly execute a program inside of a production script you must use runtime compiler options to pass the program its unit assignments, log its start and stop time, check its return code and execute appropriate

ECF hooks respective to the return code. This all sounds daunting but ***setup.sh*** will assist you in meeting these standards by gathering the needed utilities into your scratch area. After running this script, the utilities ***prep_step, err_chk, err_exit, postmsg*** and ***startmsg*** will be available for use. These five utilities are described below. You should always run setup.sh every time you change directories in a script; this will ensure the utilities listed below are available in your current working directory.

prep_step
In production, for the IBM WCOSS, the environmental variable FORTk (where k is a 2-digit number) is used to pass unit assignments to the program. Since there may be multiple Fortran programs running inside of a job, these variables must be reset before each program execution. Running ***prep_step*** before each program execution will unset all FORTk variables currently set in the environment.

postmsg
postmsg simply writes a message to a log file. The first argument is the log file name and the second argument is the message. You should use the log file named /com/logs/jlogfiles/jlogfile.${pid} (where ${pid} is the process id associated with your job and which is set in your J-job script) when using ***postmsg*** in a production job.

err_chk
The script ***err_chk*** is used to check for a non-zero return code of a program execution and run a series of commands based on this return code. If a program executes with a return code of zero the end time is logged and job execution continues. If a non-zero return code is found stdout/stderr are written to the job output log, the time of the error is logged, an abort flag is sent back to ecFlow and the job is cancelled. The return code is passed into ***err_chk*** by setting the environmental variable $err.

err_exit
The script ***err_exit*** performs the same tasks as a non-zero return code passed to ***err_chk***.

startmsg
startmsg simply posts the start time of the program to be executed to a log file. The name of the log file is set through the standard environmental variable called jlogfile. Example 2-3 shows how to use the above 5 utilities.

Example 2-3.
This example shows the typical flow of control through a set of scripts that are generally used in production. For the control of job submission NCO uses a program called ecFlow which resides on two separate nodes (production and backup) on each WCOSS system. EcFlow has a graphical- and text-based interface, and allows job submission and scheduling based on time, another job's state, and other mechanisms to control when jobs are submitted to WCOSS.

Ecf script and def files control all of the ecf triggers and submission criteria. While it is NCO's responsibility to manage the ecf scripts, it is the responsibility of the programmer to provide relevant information regarding resources needed on WCOSS, time to run, and dependencies. The detail of this information can vary greatly depending on how complex the program or model is, and how often it needs to run. This information must be included in the Technical Impact Statement on the RFC form. RFCs missing this information will be returned to the developer.

Example 2-3 shows simple job scripts that execute a Fortran program using the utilities described above. A majority of the environmental variables set are standard variables used by these production utilities as listed in Table 1-5.   To run this example script as a batch job you must use the bsub command on the IBM WCOSS. This example will create a job output file (jlogfile.${pid}) and a subdirectory in your current working directory. The output file jlogfile.${pid} is a log of the start and end times of the job and Fortran executable. The subdirectory will contain all the utilities discussed above plus the input and output files created by the script and executable.

Example 2-3 Job submission script
In production, the job submission scripts reside on the ecFlow node and may only be modified by the SPA team.   A production version of this script would normally include more variables to establish and maintain a conversation with the production ecFlow node. This example contains only basic job card information and some "standard" variables used in all jobs, as such it can be submitted (using bsub) to WCOSS manually. The ecFlow node has a mechanism to call bsub within its configuration.

```
#BSUB -J %E%wx_control2rule_the_world
#BSUB -o /com/output/%ENVIR%/today/wx_control2rule_the_world.o%J
#BSUB -e /com/output/%ENVIR%/today/wx_control2rule_the_world.o%J
#BSUB -L /bin/sh
#BSUB -q prod_serv
#BSUB -cwd /tmpnwprd
#BSUB -W 01:00
#BSUB -P %PROJ%
#BSUB -R affinity[core]
#BSUB -n 1
#BSUB -R span[ptile=1]
#BSUB -R rusage[mem=500]

%include <head.h>

# EXPORT list here
set -x
export envir=%ENVIR%
export job=`date -u +wx_control2rule_the_world_%%H%%M`
export cyc=`date -u %%H`
export ffhr=0

VERSION_FILE=/nw${envir}/versions/wx_control2rule_the_world.ver
if [ -f $VERSION_FILE ]; then
  . $VERSION_FILE
else
  ecflow_client --abort
  exit
fi
```

```
# CALL executable job script here
/nw${envir}/jobs/JWX_CONTROL2RULE_THE_WORLD

if [ $? -ne 0 ]; then
  ecflow_client --abort
  exit
fi
```

%include <tail.h>

This is the end of the job card sample; typically this calls the job script which resides on WCOSS. Note that the environment variable that determines if the job will be run in prod, para, or test was set a few lines above the J-job script.

The LSF equivalents for CCS LoadLeveler commands (e.g. total_tasks, tasks_per_node, task_affinity, etc.) may be found at http://wcossdocs.ncep.noaa.gov/userwiki/index.php/Main_Page.

It is the responsibility of the developer to determine and provide the system resource requirements in the RFC's Technical Impact Statement, including expected runtime, nodes, consumable memory, and tasks, if applicable.

Example 2-3 J-job Script Using Utilities from setup.sh
This begins the sample of the job scripts. Typically, the ecf job card (example shown above) will call the appropriate Job script (also referred to as the J-job) in /nw${envir}/jobs where ${envir} refers to an exported variable corresponding to prod, para, or test.

Job scripts can further call "model" scripts (also called ex-scripts) in the /nw${envir}/scripts directory which can call other utility or "ush" scripts in the /nw${envir}/ush directory and Fortran and/or C executable code in the /nw${envir}/exec directory.

It is important to note that the Job script should set up the entire environment for the scripts and executables that it calls through the use of exported variables. To test a job in a different environment, it should only be necessary to change the exported variables in the job script. Other scripts should not alter the COMIN, COMOUT or other file location variables established in the Job script, and those location variables should always be used in all downstream scripts.

Further, no output files should be written to a path using a location variable defined as being used for input files, as this can cause testing to overwrite and corrupt production output.

```
################################################
#   SAMPLE J-JOB SCRIPT JWX_CONTROL2RULE_THE_WORLD
################################################
set -xa
export PS4='$SECONDS + '
date
```

```
#####################################################
# Default to NCO production run environment
#####################################################
export RUN_ENVIR=${RUN_ENVIR:-prod}

################################
# Specify NET and RUN name
################################
export NET=${NET:-gfs}
export RUN=${RUN:-gfs}

###############################################################
# This block can be modified for different Production test
# environment. This is used for operational testings
###############################################################
if [ "$RUN_ENVIR" = prod -a $envir != prod ]; then
  export DBNROOT=/nwprod/spa_util/fakedbn
  export jlogfile=/com/logs/${envir}/jlogfile
  export SENDDBN=NO
fi

#####################
#Determine job output name on the system
#####################
export pid=$$

export outid="LL$job"
export jobid="${outid}.o${pid}"
export pgmout="OUTPUT.${pid}"

export DATA=${DATA:-/tmpnwprd/${job}.${pid}}
mkdir -p $DATA
cd $DATA

export cycle=${cycle:-t${cyc}z}


################################
# setup jlogfile
################################
export jlogfile=${jlogfile:-/com/logs/jlogfiles/jlogfile.${job}.${pid}}

##################################
# SENDECF  - Flag Events on ecFlow
```

```
# SENDCOM  - Copy Files From TMPDIR to $COMOUT
# SENDDBN  - Issue DBNet Client Calls
###################################
export SENDECF=${SENDECF:-YES}
export SENDCOM=${SENDCOM:-YES}
export SENDDBN=${SENDDBN:-YES}


####################################
# Specify Execution Areas
####################################
export HOMEmodel=${HOMEmodel:-/nw${envir}}
export EXECmodel=${EXECmodel:-${HOMEmodel}/exec}
export FIXmodel=${FIXmodel:-${HOMEmodel}/fix}
export USHmodel=${USHmodel:-${HOMEmodel}/ush}
export PARMmodel=${PARMmodel:-${HOMEmodel}/parm}

export PARMutil=${PARMutil:-/nwprod/util/parm}
export FIXutil=${FIXutil:-/nwprod/util/fix}
export USHutil=${USHutil:-/nwprod/util/ush}
export utilscript=${utilscript:-/nwprod/util/ush}
export utilities=${utilities:-/nwprod/util/ush}
export EXECutil=${EXECutil:-/nwprod/util/exec}

#############################
# Run setup to initialize working directory and utility scripts
#############################
${utilscript}/setup.sh

###############################
# Run setpdy and initialize PDY variables
###############################
${utilscript}/setpdy.sh
. PDY

###############################
# set COMIN, COMOUT, first guess directories
###############################
export COMIN=${COMIN:-/com/${NET}/${envir}/${RUN}.${PDY}}
export COMOUT=${COMOUT:-/com/${NET}/${envir}/${RUN}.${PDY}}
export GESDIR=${GESDIR:-/nwges/${envir}/model.${PDY}}
mkdir -p $COMOUT

env
###############################
# execute the script
${HOMEmodel}/scripts/exwx_control2rule_the_world.sh.ecf
```

```
################################

date
cd /tmpnwprd
rm -rf $DATA
```

Example 2-3 Sample ex-script exwx control2rule the world.sh.ecf

```sh
#!/bin/sh

set -x

cp $COMIN/inputfile .
startmsg
Pgm=`basename $EXECscript/wx_control2rule_the_world`
. prep_step

$EXECmodel/wx_control2rule_the_world >> $pgmout 2> errfile
err=$?
if [ "$err" -ne 0 ]; then
  msg = "my code failed"
  postmsg "$jlogfile" "$msg"
  errexit
else
  msg = "success"
  postmsg "$$jlogfile" "$msg"
fi

if $SENDCOM = YES
  if [ -s outputfile ] then
     cp outputfile $COMOUT
  fi
fi

#####################################
# GOOD RUN
Set +x
Echo "**************JOB WX_CONTROL2RULE_THE_WORLD COMPLETED NORMALLY"
Echo "**************JOB WX_CONTROL2RULE_THE_WORLD COMPLETED NORMALLY"
Echo "**************JOB WX_CONTROL2RULE_THE_WORLD COMPLETED NORMALLY"
#####################################

Msg='job completed normally.'
Echo $msg
Postmsg "${jlogfile}" "${msg}"
Exit
######## END OF SCRIPT ########
```

Conclusion

Each model is different and requires creative techniques to achieve the best forecast. It is not the intent of this document to limit creativity or squash innovation. It is necessary to establish and promote the use of common utilities, directories, and practices that lead to more efficiency when testing a change; and when troubleshooting failures. These guidelines will continue to evolve and with everyone's help become more complete in well thought out processes and best practices.