MEMORANDUM FOR: CCS Programmers
FROM:
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 CCS. The formal process to request a change or addition is a Request For Change (RFC) and is discussed in another document. The goal of this document is to discuss the need and provide example 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 CCS. 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.

d) If possible, one makefile in the top level source directory should do everything to build the executable.

e) If possible the executable produced should be named the same as the top level source directory that contains it.

f) If the make process builds more than one executable then the make process should copy the executables to the top level source directory and be sure their filenames are the same as the scripts that call them expect.

g) Where possible, simple executables that use I/O should follow standard unit number conventions as listed in attachment II. We realize this standard is antiquated and can not be followed in all cases, but when possible, it helps in troubleshooting to know what files are considered input and which are designated as output.

<u>Further discussion</u>

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 CCS, that can be accessed 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 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. See Attachment I for information regarding makefiles.

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:  AIX UNIX
#     Machine:    NCEP  CCS  p5 p6
#
```

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) at the end of the directory name.

My_global_weather_model_to_rule_the_world.fd would therefore contain Fortran code and a makefile to produce the My_global_weather_model_to_rule_the_world executable which would be created in the My_global_weather_model_to_rule_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 change package. Detailed instructions should be provided to NCO, instructing them as to what code in the change 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 = nam_combc # The name of the executable produced #
#
# CPP, Compiler, and Linker Options
FC       = ncepmpxlf # Fortran compiler used #
CPP      = /lib/cpp -P
ARCH     = auto
CPPFLAGS =
# Optional compiler directives
OPTS     = -qnosave -qarch=$(ARCH) -qmaxmem=-1 -NS2000
LIST     =
FREE     =
FIXED    = -qfixed
TRAPS    =
PROFILE  =
DEBUG    = -g
MEM      =
MAP      = -bloadmap:mapfile
W3LIBDIR = /nwprod/lib
ESSL     = -lessl
MASS     = -lmass

# 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     = $(ESSL) $(SEARCH) $(NCDLIBS) -L$(W3LIBDIR) -lw3_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 what 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 CCS, users should us the environmental variable **XLFUNIT_#**.

Example:

```
export XLFUNIT_16="inputfilename"
export XLFUNIT_60="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 it is being changed. To ensure that the latest version of the script is used, programmers should modify the operational scripts which they have copied from /nwprod.
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 POSIX Shell (/bin/sh).
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 $pgmout (standard production variable).
i. In each system of scripts, the top level script creates and initializes the standard production environmental variables and resides in /nwprod/jobs with the standard naming convention of JXXXX.sms.prod. The top level J-job script called the main driver script which resides in /nwprod/scripts with the standard naming convention of exXXX.sh.sms. Any needed sub-scripts to the main driver script will be located in /nwprod/ush or /nwprod/util/ush.
j. Production utilizes a centralized cleanup and creation of directories in /com and /nwges. Production scripts should not remove or create production directories at the /com/$NET/$envir/$RUN.$PDY level.

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;

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.

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.

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.

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.

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
/dcom	Incoming Data
/pcom	Outgoing Products

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*.*YYYYMMDD*	Production Spin-up data for model
test/*model_name*.*YYYYMMDD*	Test Spin-up data for model
para/*model_name*.*YYYYMMDD*	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_name*.*YYYYMMDD*	Production Model Output for a day
model_name/test/*net_name*.*YYYYMMDD*	Test Model Output for a day
model_name/para/*net_name*.*YYYYMMDD*	Parallel Model Output for a day
output/prod/*YYYYMMDD*	Job stdout/stderr for a day
output/test/*YYYYMMDD*	Job stdout/stderr for a day
output/para/*YYYYMMDD*	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 Scratch Directory
jlogfile	Logfile of start and end time of all jobs
outid	Job id appearing in jlogfile
jobid	Jobid appearing in scratch directory name
pgmout	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
SENDSMS	Enable/Disable SMS 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 files
USH*model_name*	Directory containing supporting model scripts
SMSBIN	Directory containing SMS executables

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

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 Script Using finddate.sh

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

today=20020101

# 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 tens days will be $ten_days_ahead”
echo
echo “The last four days where $last_four_days”
echo “The next four days are $next_four_days”
```

Example 2-1 Output

```
Today’s Date is 20020101

The date ten days ago was 20011222
The date in tens days will be 20020111
```

The last four days where 20011231 20011230 20011229 20011228
The next four days are 20020102 20020103 20020104 20020105

setpdy.sh

setpdy.sh is a shell script to help 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 days date. If the environmental variable PDY is set when executed, the centered date will be the value of PDY.

This utility script uses date files in /com/date set by 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 days 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.

Example 2-2 Script Using setpdy.sh

```
#!/bin/sh
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=20020101
export cycle=t12z
$utilscript/setpdy.sh
. PDY
```

Example 2-2 Contents of File

```
Export PDYm7=20011225
export PDYm6=20011226
export PDYm5=20011227
export PDYm4=20011228
export PDYm3=20011229
export PDYm2=20011230
export PDYm1=20011231
export PDY=20020101
export PDYp1=20020102
export PDYp2=20020103
export PDYp3=20020104
export PDYp4=20020105
export PDYp5=20020106
export PDYp6=20020107
export PDYp7=20020108
```

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 SMS 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 you must use the runtime compiler options and variables to pass a Fortran program its unit assignments. For the IBM SP, the environmental variable XLFUNIT_*numberi* 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 set the variable XLFRTEOPTS to enable the use of the variable XLFUNIT_*number*, and will unset all XLFUNIT_*number* 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/jlogfile 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 SMS and the job is cancelled. The return code is passed into ***err_chk*** by setting the environmental variable err.

err_exit

The script 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.

This example shows the typical (well simplistic) 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 SMS which resides on a separate computer system than the CCS. It 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 the CCS.

Sms script and def files control all of the sms triggers and submission criteria. While it is NCO's responsibility to manage the sms scripts, it is the responsibility of the programmer to provide relevant information regarding resources needed on the CCS, time to run, and dependencies. The detail of that information can depend greatly on how complex the program or model is, and how often it needs to run.

This example 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 yon must use the llsubmit command on the IBM SP. This example will create a job output file, jlogfile and subdirectory in your current working directory. The file called jlogfile is a log of the start and end times of the job and Fortran executable. Inside of the subdirectory will be all the utilities discussed above plus the input and output files created by the script and executable.

Job submission script – this would normally include more variables to establish and maintain a conversation with the sms computer. This example contains only basic job card information and some "standard" variables used in most all jobs, as such it can be submitted (using llsubmit) to the CCS manually. The sms computer has a mechanism to call llsubmit within its configuration.

```
# @ job_name = jweather_control2rule_the_world
# @ output = /com/output/test/today/weather_control2rule_the_world.o$(jobid)
# @ error = /com/output/test/today/weather_control2rule_the_world.o$(jobid)
# @ shell = /bin/sh
# @ wall_clock_limit = 00:15:00
# @ class = prod
# @ job_type = parallel
# @ network.MPI = sn_all,shared,us
# @ initialdir = /tmpnwprd
# @ notification = never
# @ account_no = IBM001-ADM
# @ resources = ConsumableMemory(300 MB)
# @ task_affinity = cpu(1)
# @ parallel_threads = 1
# @ queue

echo $LOADL_PROCESSOR_LIST
export MP_SHARED_MEMORY=yes
export MEMORY_AFFINITY=MCM

# EXPORT list here
```

```
export envir=test
export job=weather_control2rule_the_world
export cyc=00
export ffhr=0

/nw${envir}/jobs/JWX_CTRL2RULEWORLD.sms.test
```

The end of the job card sample; typically this calls the job script and sets the environment variable that determines if the job will be run in prod, para, or test.

Example 2-3 Script Using Utilities from setup.sh

This begins the sample of the job scripts. Typically, the SMS job card will call the appropriate Job script in /nw${envir}/jobs where ${envir} refers to an exported variable corresponding to prod, para, or test.

Job scripts can further call “model” scripts in /nw${envir}/scripts directory which can call other “ush” scripts in the /nw${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 all of the 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.

```
#!/bin/sh
set -xa
export PS4='$SECONDS + '

export pid=$$

export outid="example2-3"
export jobid="${outid}.o${pid}"
export pgmout="OUTPUT.${pid}"
#
# Set cycle time for setpdy.sh to figure out the date
#
export cyc=12
export cycle=t${cyc}z
#
# Set Name of stdout/stderr file for executables
#
export pgmout="OUTPUT.$$"
#
# Set Temporary Scratch Directory
#
export DATA=/tmpnwprd/${jobid}
mkdir -p ${DATA}
```

```
#
# Run inside of Scratch Directory
#
cd ${DATA}
#
# Disable SMS Hooks
#
export SENDSMS=NO
#
# File to log job information
#
export jlogfile=../jlogfile
#
# Run setup.sh to copy utilities into scratch directory
#
export utilscript=/nwprod/util/ush
sh $utilscript/setup.sh

msg="Beginning of Example Job Sucessfully"
postmsg "$jlogfile" "$msg"
#
# Set the Date Variables
#
sh $utilscript/setpdy.sh
. PDY
#
# Try running with the line below commented out to change the
# outcome of the script
#
echo $PDY > date_file_in
#
# Set Standard Variable pgm for use by utilities then
# source prep_step to set XLFRTEOPTS and unset XLFUNIT vars
#
export pgm=example2-3
. prep_step

#
# Log start of program execution
#
startmsg
../example2-3 >> $pgmout 2>errfile
#
# Check the return code of example2-3
#
export err=$?;err_chk

msg="Got to End of Example Job Sucessfully"
postmsg "$jlogfile" "$msg"

exit
```

Example 2-3 Fortran source code

```
      program example2-3

! $$$$ MAIN PROGRAM DOCUMENTATION BLOCK
!
! Main Program: example2-3
!  Prgmmr: Mabe
!
! Abstract: Sample code for Implementation Standards Document
!
! History Log:
!   09-03-10 I made this today
!
! Usage: main
!  Input Argument List:
!    None
!  Output Argument List:
!    None
! Subprograms called:
!  Utilities:
!  Library:
!
! Attributes:
!  Language: Fortran 90
!  Machine: IBM RS6000 SP
!
! $$$$
       implicit none

       write ( *,'(a)' ) ' '
       write ( *,'(a)' ) 'This is a test'
       write ( *,'(a)' ) 'Hello World! '

       stop
end
```

Obviously this is a very simplistic example program only to illustrate the documentation requirement. The documentation block does not have to be formatted in exactly this way, however, it should be as descriptive as is practical and present.

Example 2-3 Sample Output of jlogfile with Successful and Unsuccessful Runs

01/24 22:29:21Z example2-3.o21036-Beginning of Example Job Successfully
01/24 22:29:22Z example2-3.o21036-example2-3 started
01/24 22:29:22Z example2-3.o21036-example2-3 completed normally
01/24 22:29:22Z example2-3.o21036-Got to End of Example Job Successfully

01/24 22:30:21Z example2-3.o24088-Beginning of Example Job Successfully
01/24 22:30:22Z example2-3.o24088-example2-3 started
01/24 22:30:22Z example2-3.o24088-example2-3 started
01/24 22:30:22Z example2-3.o24088- FAILED example2-3.o24088 – ABNORMAL EXIT

There is some discussion about the continued use of the jlogfile. Certainly, as the system gets bigger and busier, many programs that write to a single text file can become contentious, however, for now we continue to support messages to the jlog. However, the more sparse these writes are, the better.

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.