LAMPS 90

USER'S GUIDE

August 11, 1994

Robert Cohen

Numerical Model Development Facility

Drexel University

NOTICE

LAMPS 90 is written by and for the Numerical Model Development Facility at Drexel University. For further information contact Drs. Donald Perkey and Carl Kreitzberg at Drexel University, Department of Physics and Atmospheric Science, Philadelphia, PA, 19104.

Formatting Convention Used in This Document

word type	style	examples
files and directories	**bold**	**src/ case.cfg merge.F**
UNIX or FORTRAN commands	`typeface`	`make #include read`
parameters	*emphasis*	*prog case*

Contents

1 INTRODUCTION TO LAMPS 90 **8**
- 1.1 Short description of LAMPS 90 . . . 8
- 1.2 Short description of this documentation . . . 8
- 1.3 Note to novice users . . . 8
- 1.4 Prerequisites . . . 9
- 1.5 Tutorial . . . 9
- 1.6 History . . . 10
- 1.7 Using this documentation . . . 10

2 STRUCTURE OVERVIEW **11**
- 2.1 Directory Structure . . . 11
  - 2.1.1 MAIN LAMPS 90 DIRECTORY . . . 11
  - 2.1.2 User LAMPS 90 Directory . . . 12
  - 2.1.3 Case Directory . . . 12
- 2.2 Using LAMPS 90 PROGRAMS . . . 13
  - 2.2.1 Compiling LAMPS 90 PROGRAMS . . . 13
  - 2.2.2 Running LAMPS 90 PROGRAMS . . . 13

3 INSTALLATION PROCEDURES **15**
- 3.1 Getting the LAMPS 90 package onto your machine . . . 15
- 3.2 Configuring the makefiles . . . 15
  - 3.2.1 Definition files . . . 16
  - 3.2.2 Suffix-conversion files . . . 16
- 3.3 Machine-dependent parameters . . . 18
- 3.4 Using `Install` . . . 18
  - 3.4.1 Makefile changes . . . 19
- 3.5 Changes to the SOURCE CODE . . . 19

4 CASE SET-UP **20**
- 4.1 Running Setup user . . . 20
- 4.2 Running Setup case . . . 20
- 4.3 Getting DATA FILES . . . 21
  - 4.3.1 Case-independent data . . . 21
  - 4.3.2 Case-dependent data . . . 21
- 4.4 INPUT FILES . . . 21
  - 4.4.1 Using the sample INPUT FILES . . . 22
  - 4.4.2 Running Setup input . . . 22

5 COMPILING AND RUNNING PROGRAMS **23**
- 5.1 Compiling PROGRAMS . . . 23
  - 5.1.1 Using your own source code . . . 24
  - 5.1.2 Using your own libraries . . . 24
- 5.2 Running PROGRAMS . . . 25

5.2.1 Basic run . . . 25
5.2.2 Normal run . . . 25
5.2.3 Forced run . . . 26
5.2.4 Using other log, input or graphics sub-directories . . . 26
5.2.5 Miscellaneous information . . . 26
5.3 Makefile Structure . . . 26
5.4 Short-cuts for multiple-PROGRAM procedures . . . 27

6 PROGRAMS 29
6.1 INITIALIZATION . . . 29
6.1.1 Real Data . . . 29
6.1.2 Analytic Data . . . 31
6.2 Dynamic Model . . . 31
6.2.1 Full Physics . . . 32
6.2.2 Dry Simple . . . 32
6.3 Post-processing . . . 32
6.3.1 Accessing Data Volume Information . . . 33
6.3.2 Reformatting a History Data Volume . . . 33
6.3.3 Graphics . . . 33
6.3.4 Trajectories . . . 34

A Glossary 35

B Fixed recordlength, unformatted files 37

C Real number definitions 38

D DATA FILES 39
D.1 Input Cards . . . 39
D.2 Database Files . . . 40
D.3 Log Files . . . 40
D.4 Data Volume . . . 40
D.4.1 Terrain Data Volume . . . 41
D.4.2 Initialization Data Volume . . . 41
D.4.3 History Data Volume . . . 42
D.5 Model Initial Data File . . . 43
D.6 Time-dependent Boundary File . . . 43
D.7 Trajectory Listings . . . 43
D.8 GMETA Files . . . 43

E COMPUTER-SPECIFIC PECULIARITIES 44
E.1 Sun 4 / Sparcstation 1+ . . . 44
E.2 CONVEX C1 . . . 45
E.3 Stardent . . . 45
E.4 MSFC/CRAY . . . 46
E.5 Sun 3 / SUNOS v4.0 / F77 v1.2 . . . 47
E.6 Sony/NEWS . . . 47
E.7 POPS/CRAY . . . 48
E.8 HP-UX 9000 . . . 50
E.9 IBM AIX RS-6000 . . . 51
E.10 Silicon Graphics / IRIX . . . 52

F PREPROCESSOR LANGUAGE **53**
F.1 C-preprocessor (cpp/F) . . . 53
F.2 presplit-preprocessor (pre) . . . 54
F.3 lnwrap-preprocessor . . . 54
F.4 ifmac-preprocessor (im) . . . 54
F.4.1 IFTRAN / FORTRAN differences . . . 54
F.4.2 IFMAC / CPP differences . . . 55
F.5 Converting from IFMAC to pre . . . 55
F.6 Converting from pre to ifmac . . . 56

G PROGRAMMING PHILOSOPHY **57**
G.1 Files . . . 57
G.2 Programs . . . 57
G.2.1 Source Code . . . 57
G.2.2 Input Files . . . 58
G.2.3 Include Files . . . 58
G.3 Libraries . . . 59
G.4 Names of common variables . . . 59

List of Tables

2.1 Sub-directories of the main LAMPS 90 directory 11
2.2 Sub-directories of the case directory 13

3.1 Definition files for various machine 16
3.2 Definition file macros 16
3.3 List of suffix files 17
3.4 Machine-dependent macros 18

5.1 List of make short-cuts 27

6.1 Data files needed for the initialization process 31

D.1 Summary of database files 40
D.2 Block identification numbers in the initialization data volume 41

List of Figures

3.1 Flowchart for determining the appropriate suffix file for your machine 17

6.1 Flowchart depicting the sequence of programs needed to create an initial data set with real data. The **dvutil** program is normally used although it is not necessary. All of the programs after **begin** require the output from **begin**. It is also possible to by-pass the **reha3d**, **masa3d** and/or **wina3d** programs and have the **merge** program use the **fgsa3d** output directly. . . . 30

Chapter 1

INTRODUCTION TO LAMPS 90

Welcome to the Limited-Area, Mesoscale Prediction System for the 1990's (LAMPS 90).

1.1 Short description of LAMPS 90

LAMPS 90 is a set of PROGRAMS[1] that take an initial state of the atmosphere (derived from observations or analytic conditions), simulate the evolution of the atmosphere from the initial state and graph and analyze the results. Each PROGRAM performs one step of the process.

The PROGRAMS and all of the files associated with the compilation and running of the PROGRAMS are organized and arranged in a structure that takes advantage of the capabilities of a UNIX machine and makes the process of installing, compiling and running the PROGRAMS relatively straightforward.

1.2 Short description of this documentation

This documentation describes the structure of LAMPS 90 and explains how one goes about installing, compiling and running the LAMPS 90 system of PROGRAMS. For information regarding a particular PROGRAM, see the *LAMPS 90 Programmer's Guide*.

1.3 Note to novice users

As mentioned above, the world of numerical modeling and LAMPS 90 consists of various parts. To the novice it probably appears a little intimidating and overwhelming, but by identifying the parts it should become navigatable. Please be patient and continue reading.

LAMPS 90 contains many different programs. This is many different programs are needed in order to do all of things that are needed to do. There are three types of programs: INITIALIZATION, DYNAMIC MODEL, POST-PROCESSING.

The principle part of LAMPS 90 is the DYNAMIC MODEL. This is the PROGRAM that makes a forecast given

[1] All words written in SMALL CAPS type are defined in the glossary, appendix A

an initial field. Contrary to what one might expect, the DYNAMIC MODEL is not just a bunch of equations processed by the computer to spit out a forecast. In addition to the equations, there is a significant amount of code needed to handle the I/O (i.e., input and output) of the DYNAMIC MODEL and memory limitations of the computer. Not too surprisingly, several different files are required to contain all of the source code for the DYNAMIC MODEL PROGRAM.

Furthermore, LAMPS 90 does not just consist of a DYNAMIC MODEL. There are also PROGRAMS to process the observations (for interpolating to model grid and removing errors) and PROGRAMS to process the forecasts (for graphing). The former are called INITIALIZATION PROGRAMS (because they create an initial data field) and the latter are called POST-PROCESSING PROGRAMS (because they are used after the forecast has been made).

The source code for the LAMPS 90 programs can be quite complicated. In addition, it can be quite difficult, especially for the novice, to find all of the source code for a particular PROGRAM. An attempt has been made to address this problem by providing a clear file structure in LAMPS 90 (see chapter 2). In addition, procedures have been laid out for installing LAMPS 90 (chapter 3) and compiling and running programs (chapter 5).

Still, everybody is different. Some users just need to run the programs. Others need to make changes to the programs or write additional programs. While attempts have been made to accommodate both, LAMPS 90 is geared more for the latter. As such, certain structures may appear overly complicated to the novice.

1.4 Prerequisites

Before one attempts to use LAMPS 90, it is strongly recommended that the user become familiar the computer operating sysmte (i.e., UNIX), the NCAR GRAPHICS library (for plotting any graphs that are made) and an editor (for editing files). This documentation will assume that the user is familiar with these things. If you are unfamiliar with any of these, it is suggested that you read the following files.

UNIX	read **doc/help/unix.help**
vi editor	read **doc/help/vi.help**
NCAR GRAPHICS	consult your NCAR GRAPHICS user manual

Note: Be aware that each machine might have its own way of handling NCAR GRAPHICS. Usually, one needs to enter a 'plot' mode, through commands such as `plt`, `ictrans` or `idt`.

Note: There are two additional help files available, **doc/help/make.help** and **doc/help/sh.help**. If you are planning to do programming or any work with the makefiles but lack experience in either using `make` or `sh` , it may be helpful to look at these help files.

1.5 Tutorial

A tutorial is available that guides the user through the steps needed to install, compile and run a sample of LAMPS 90 PROGRAMS. The tutorial is included as part of the LAMPS 90 package as the **doc/tutorial** file. The tutorial is not in POSTSCRIPT and can be printed out on any printer.

This tutorial is self-sufficient and allows users to experience the LAMPS 90 environment without having to read through all the documentation. Very little information is given on how LAMPS 90 is structured, however. Such information is critical for the successful use of LAMPS 90. That is, one needs to know how

to set up a new forecast, where to specify to the PROGRAMS what you want them to do, where the files are stored, where and how to compile and run a PROGRAM, and how to check that the PROGRAM ran normally. If you find this information lacking in the tutorial, read chapter 2 (Structure Overview) to learn more about these kinds of details.

1.6 History

LAMPS 90 is based upon LAMPS, a package created at Drexel University in the 1970's. For those familiar with the old LAMPS, LAMPS 90 differs from the old LAMPS in three principal ways.

1. The first is language. The old LAMPS was written in IFMAC, a language that requires a preprocessor to convert the code into FORTRAN 77. LAMPS 90 is written in FORTRAN 77 with C-preprocessor extensions and an internal subroutine preprocessor (i.e., **presplit**). It is expected that with FORTRAN 90, many of these extensions will become obsolete.
2. The second difference is job control. LAMPS 90 utilizes the makefile structure present on UNIX machines to aid in compilation, linking and running. In addition, LAMPS 90 takes advantage of the UNIX directory structure in order to organize the LAMPS 90 files. These changes make the package best for UNIX machines and, although it is not restricted in this way, this documentation assumes that the user is on a UNIX machine.
3. The third is the DATA VOLUME. The old LAMPS used a DATA VOLUME that was machine-dependent, with the packing scheme set by each call to the I/O routines. LAMPS 90 utilizes a DATA VOLUME that is independent of machine, case and record length. In addition, the output from the DYNAMIC MODEL (i.e., the history) is now in the DATA VOLUME format.

For more information on the changes which have been made, see the **doc/updates.doc** file.

1.7 Using this documentation

It is suggested that the novice user first read chapter 2 (Structure Overview) which will review the basic components of LAMPS 90 and also introduce the terminology. Terms that are defined in the glossary are displayed in SMALL CAPS.

Depending on your objective, your next step should be as follows.

- To learn more about installing LAMPS on your machine,
  - ... read chapter 3 (Installation Procedures) and
  - ... read chapter 4 (Case Set-up) and
- To compile and run the PROGRAMS,
  - ... read chapter 5 (Compiling and Running Programs).
- To learn what PROGRAMS are available,
  - ... read chapter 6 (Programs).

For more detailed questions regarding the PROGRAMS, see the *LAMPS 90 Programmer's Guide.*

Chapter 2

STRUCTURE OVERVIEW

Before entering the world of numerical modeling and LAMPS 90, it is best that one familiarize oneself with the structure that will be encountered. The structure is pretty much irrelevant to the PROGRAMS, but it helps to organize everything. By becoming familiar with the structure, the user can concentrate on the PROGRAMS without being confused by purely structural novelties.

2.1 Directory Structure

In order to organize the files that are needed for LAMPS 90, a special directory structure is used. This structure may initially seem complicated. Once you get used to it, however, it will prove its usefulness.

2.1.1 MAIN LAMPS 90 DIRECTORY

The MAIN LAMPS 90 DIRECTORY contains all of the user-independent source code and makefiles needed to run LAMPS 90. The directory has five sub-directories (see table 2.1).

Of these directories, the **src/**, **config/** and **/doc** directories are further broken down. The **doc/** directory has two sub-directories, one for the help files and one for the postscript versions of the documentation. The tutorial and list-of-updates files are not stored in a separate sub-directory.

The **config/** directory has seven sub-directories.

bin/	contains codes used to compile the SOURCE CODES
config/	contains files used to setup LAMPS 90 for the user
doc/	contains documentation for LAMPS 90
incl/	contains files to be included by the source code
lst/	contains lists
src/	contains the SOURCE CODE for all the LAMPS 90 PROGRAMS

Table 2.1: Sub-directories of the main LAMPS 90 directory

defs/	machine-dependent macro definitions (for the makefiles)
incl/	include-type files that depend upon machine and case
inp/	files containing sample input commands for each PROGRAM
mks/	program-dependent makefiles
setup/	files used to set-up each user and case
sfx/	instructions for machine-dependent compiling (for the makefiles)
src/	source codes that are machine-dependent

Most of these programs are utilized when the user configures a directory so that LAMPS 90 can be used efficiently. The user, however, is usually unaware that these files are being accessed because the configuration is done mostly via a series of shell commands (see chapter 4).

The **src/** sub-directory is broken down into four more sub-directories.

init/	contains INITIALIZATION codes
model/	contains DYNAMIC MODEL codes
post/	contains POST-PROCESSING codes
rlib/	contains codes for libraries

Each of these sub-directories of **src/** and then further broken down such that each PROGRAM is contained in a separate directory. For example, there is a sub-directory called **src/init/frstg25/** that contains all the SOURCE CODE for providing the first guess field from the NMC 2.5 degree data (part of the INITIALIZATION process). Likewise, all of the routines for the library **liblamps** are contained in the directory **src/rlib/liblamps**.

Additional sub-directories can be created to store user-independent files like databases (e.g., **dbs/**).

All of the files and directories within the MAIN LAMPS 90 DIRECTORY **should remain untouched by the users. Changes to these files and directories should be carried out only with the approval of all users.** In general, once LAMPS 90 is installed, the entire MAIN LAMPS 90 DIRECTORY can be made write protected. The only exception is the **bin/** sub-directory, which contains various programs that may need to be compiled by the various users.

2.1.2 User LAMPS 90 Directory

Each user creates his/her own LAMPS 90 environment, separate from the MAIN LAMPS 90 DIRECTORY, in which to carry out experiments and such. Each user accesses the MAIN LAMPS 90 DIRECTORY, but all the work is performed within each user's individual LAMPS 90 environment. In this way, other users remain unaffected. This structure also allows the user to run the PROGRAMS and create DATA FILES on a separate disk from the MAIN LAMPS 90 DIRECTORY. This is useful for those cases that require a lot of disk space. For more information, see chapter 4 (Case Set-up).

To set up the user's individual LAMPS 90 directory, one needs to run **Setup** within the MAIN LAMPS 90 DIRECTORY (i.e., `Setup user`). This process will create the user directory (if not already present) and place within it a file called **Setup** and a file called **user.cfg**. For more information, see section 4.1 (Running setup user).

2.1.3 Case Directory

For each case (i.e., grid) to be run, the user runs **Setup** again, this time from the user LAMPS 90 directory and with the argument `case` instead of `user`. As a result of the `Setup case` process, the user's individual

case/bin/	contains EXECUTABLES for each LAMPS 90 PROGRAM
case/lib/	contains case-dependent libraries
case/incl/	contains the case-dependent include files
case/mks/	contains the case-dependent makefiles
case/obj/	contains the OBJECT CODES for the case

Table 2.2: Sub-directories of the case directory

LAMPS 90 directory will automatically contain a case sub-directory[1]. Within this CASE DIRECTORY, the user runs cases and creates DATA FILES. For more information, see section 4.2 (Running setup case).

The CASE DIRECTORY contains all of the case-dependent files. Case-independent files should be stored within the user LAMPS 90 directory rather than the within the CASE DIRECTORY. If not already present, the user's individual LAMPS 90 directory will also contain a **lib/** sub-directory (for case-independent libraries).

The setup process will automatically create the necessary files and directory structure for the user to run a case. As a result, the CASE DIRECTORY will automatically contain five sub-directories (see table 2.2) and a makefile called **Makefile**.

Additional sub-directories can be created to store case-dependent source codes (e.g., **src/**), data (e.g., **data/**) and lists (e.g., **lst/**).

2.2 Using LAMPS 90 PROGRAMS

To run a PROGRAM, the user needs to compile the necessary code and then run the EXECUTABLE which was created by the compilation.

2.2.1 Compiling LAMPS 90 PROGRAMS

Since each EXECUTABLE may be dependent upon many different SOURCE CODES and libraries, it can be difficult to remember which libaries and SOURCE CODES are really needed for each EXECUTABLE. To handle all the details, a makefile structure was set up. To create an EXECUTABLE for a PROGRAM, simply type

```
make prog
```

within the user's CASE DIRECTORY (where *prog* is the name of the EXECUTABLE). As long as the user's CASE DIRECTORY has been set-up properly, this command will automatically compile all the necessary codes and produce an EXECUTABLE named *prog* within the **case/bin/** directory. See sections 4.2 (Running setup case) and 5.1 (Compiling PROGRAMS) for more information.

2.2.2 Running LAMPS 90 PROGRAMS

Each PROGRAM can be started by typing the name of the EXECUTABLE. Since

```
make prog
```

[1] Because everything related to a specific case is kept in this separate directory, this directory is called the CASE DIRECTORY. At times, this documentation will refer to the CASE DIRECTORY as *case/*. However, the name of this directory is not limited to "case".

creates an EXECUTABLE named **bin/***prog* within the user's CASE DIRECTORY, the user can start the PROGRAM by simply typing

`bin/`*prog*

within the user's CASE DIRECTORY.

Each PROGRAM, however, has a lot of options regarding the type of data, the date, the grid size, etc. All of this information has to be given to the PROGRAM and each PROGRAM expects this information to be given to it from standard input (i.e., from the screen) in the form of a prespecified command line structure. This command line structure has the following format.

.command parameter parameter parameter ...

Each parameter can be separated by any number of spaces or tabs. The '#' character can be used as a comment delimiter.

Unfortunately, there is usually no guide listing the possible COMMAND LINES for each PROGRAM[2]. Instead, the user has to depend on the SOURCE CODE for information on the various command lines acceptable for each PROGRAM.

To deal with this problem, sample INPUT FILES have been created for each PROGRAM. These sample INPUT FILES are stored in the **/config/inp/** sub-directory of the MAIN LAMPS 90 DIRECTORY. Each INPUT FILE has a list of commands usually used with each PROGRAM.

Rather than typing the COMMAND LINES directly, it is easier for the user to make a copy of the sample INPUT FILE, edit it and then redirect the input as follows.

`bin/`*prog* $<$ *prog*`.inp`

The less-than sign ($<$) is a unix way of directing the EXECUTABLE to take the input from the INPUT FILE rather than from the screen. For more information on these INPUT FILES, see sections 4.4 (INPUT FILES) and the LAMPS 90 Programmer's Guide.

For information on how to use the makefile structure to run PROGRAMS, see section 5.2 (Running PROGRAMS).

[2]some PROGRAMS allow for a .help command which lists the possible COMMAND LINES

Chapter 3

INSTALLATION PROCEDURES

LAMPS 90 is configured such that, unless you are using a completely new machine, installing the LAMPS 90 package is relatively simple. Most of the hassle will be in getting the appropriate observation files necessary to run your case. At most, there will be only a few changes which need to be done to the files.

3.1 Getting the LAMPS 90 package onto your machine

- Copy **lamps90.tar** to your system.

  A tar file named **lamps90.tar** is available which contains all of the LAMPS 90 files. The file can be obtained via anonymous ftp to nimbus.physics.drexel.edu (in sub-directory `pub/lamps90`). Contact the Numerical Model Development Facility at Drexel University[1] for more information. Within `ftp`, one can transport the tar file to your machine (specify `binary` before `get`-ing). Put the file in the directory that is to act as your MAIN LAMPS 90 DIRECTORY.

- Extract the LAMPS files from **lamps90.tar**.

  On most UNIX machines, `tar xvf lamps90.tar` will extract all the files. If this doesn't work, check the computer-specific peculiarities in appendix E. Some machines have problems with user definitions and foreign tar files.

  Once all the files have been extracted, you will notice a bunch of sub-directories. Most of these will be of no concern to the novice user.

3.2 Configuring the makefiles

By using `Setup` (see sections 4.1 and 4.2), most of the makefile structure can be automatically set up. However, there are two files which need to be supplied by the user : the **.def** file and the **.sfx** file. These files are stored in the **config/defs/** and **config/sfx** directories.

These files have already been created for each of the machines on which LAMPS 90 has been implemented. If you are using the exact same machine as one that has already been used, no changes will be necessary (although one should still double-check just to make sure). If you are not, they should be used as a guide.

[1] cwk@coasmail.physics.drexel.edu

Machine	*.def file*
sun/sparc	**sun.def**
convex/c1	**convex.def**
POPS/CRAY	**cray.def**
HPUX/9000	**hpux.def**
SONY/NEWS	**sony.def**
stardent/atmos	**stardent.def**
IBM/RS-6000	**rs.def**
SGI/IRIX	**sgi.def**

Table 3.1: Definition files for various machine

NCARG	directory in which NCAR GRAPHICS library are kept
CPP	name of C preprocessor
CC	name of C compiler
CFLAGS	flags to use with C compiler
ARCHIVE	command to create an archive
RANLIB	command to convert archives to random libraries
F77	name of FORTRAN compiler
FFLAGS	flags to use with FORTRAN compiler
F77_LD	name of FORTRAN linker
SFX_LIST	suffix conversion list (see section 3.2.2)

Table 3.2: Definition file macros

3.2.1 Definition files

The definition files have a **.def** suffix. These files contain the machine-dependent definitions used by the makefile structure (see table 3.2). As such, each machine should have its own definition file. Currently, seven definition files have been created, one for each machine currently able to run LAMPS 90 (see table 3.1).

Things to keep in mind :

- Make sure that the pathnames are correct (for NCAR GRAPHICS library, shells, etc.). You may still have to edit the **.def** files even if your machine is the same type because these pathnames depend on your machine's particular directory structure.
- Some of the code may assume that variables within subroutines are saved, even though the SAVE statement is not used. You may need to find the fortran flag that makes SAVE the default during compilation.
- If there is a FORTRAN flag that allows line lengths greater than 132, use it.

3.2.2 Suffix-conversion files

As mentioned in the previous section, one of the macros that needs to be set within the definition file is SFX_LIST, which specifies the appropriate suffix conversion file. The suffix conversion files have a **.sfx** suffix. These files contain the instructions for the makefiles regarding how compilation should be done.

At present there are only three different types. Chances are you can use one of the three already provided. They are reviewed in table 3.3. Once the appropriate suffix file has been found, this information needs to be added to the appropriate definition file (see SFX_LIST macro).

file name	machine capabilities
autoF.sfx	1. has no limit on line length line (or, at least up to 1024 characters) 2. FORTRAN compiler can handle C-preprocessor commands (i.e. it can convert directly from **.F** to **.o**).
needcpp.sfx	1. has no limit on line length line (or, at least up to 1024 characters) 2. FORTRAN compiler can't convert directly from **.F** to **.o**.
needwrap.sfx	1. has a limit on line length

Table 3.3: List of suffix files

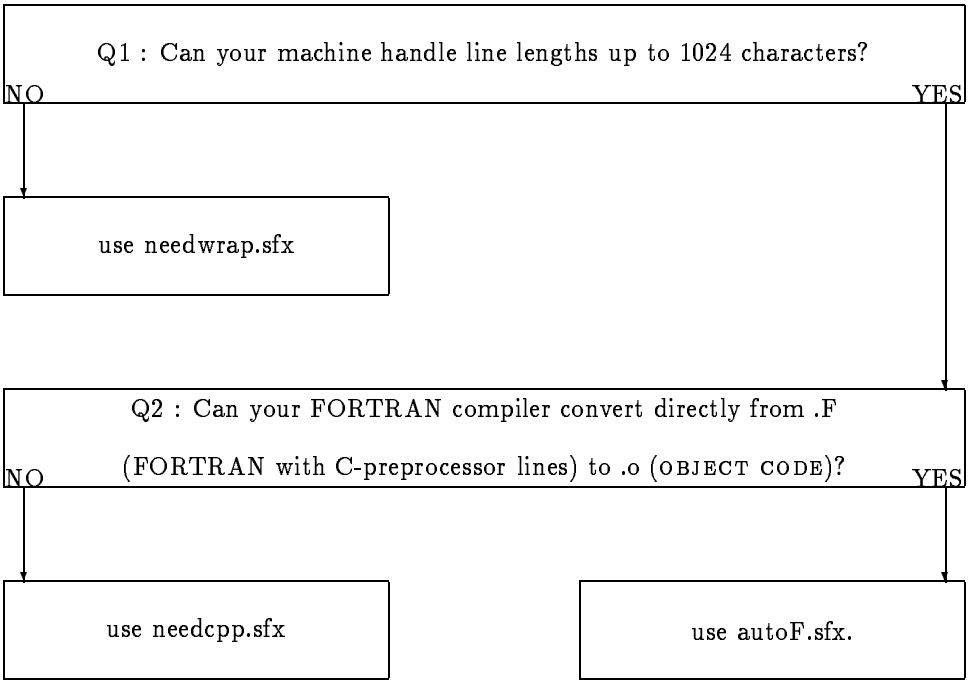

Figure 3.1: Flowchart for determining the appropriate suffix file for your machine

To determine which suffix-conversion file is appropriate for your machine, use the flowchart in figure 3.1.

For example,

Machine	Answer to Q1	Answer to Q2	`.sfx` file
sun/sparc	YES	YES	**autoF.sfx**
convex/c1	YES	NO	**needcpp.sfx**
POPS/CRAY	NO	YES	**needwrap.sfx**
HPUX/9000	YES	YES	**autoF.sfx**
SONY/NEWS	NO	YES	**needwrap.sfx**
stardent/atmos	YES	YES	**autoF.sfx**
IBM/RS-6000	MAYBE	NO	**needwrap.sfx**
SGI/IRIX	YES	YES	**autoF.sfx**

Note: If a machine can't handle the call to `getarg` within **presplit.F**, then you will need to specify **nogetarg.sfx** as `SFX_LIST` in the **.def** file and you will need to define `_noextra_` within **presplit.F**. The **nogetarg.sfx** file will do the following.

lword	the computer long word bit size
nbpwd	number of bytes per word
realmax	maximum real number
realmin	minimum real number
realfrac	minimum real fraction
oneplus	definition of 1 plus one bit
itmax	# of binary bits in the mantissa of a floating point number and is the maximum number of iterations
nb4open	# of bytes per record length designate in open call (ie. use RECL=[# of bytes per record]/_nb4open_)

Table 3.4: Machine-dependent macros

.

a. copy the **.pre** file to a temporary 'input' file

b. use **presplit** with this temporary 'input' file, creating another temporary 'output' file

c. mv the temporary 'output' file to the **.F** file

So far, no machine has been found which requires this step, but the capability is made available just in case.

3.3 Machine-dependent parameters

Since each machine has different word lengths and such, a file has been set up that specifies these parameters for each machine. This file is **incl/generic.cfg**. The **.cfg** suffix is used to identify the file as containing C-preprocessor macro definitions. This suffix is used because the file will be used to "configure" the source code (this made more sense in the days before LAMPS 90 but was kept for continuity reasons).

You do not need to change this file unless the machine is not listed in appendix E. If it is a new machine, then the first thing you need to do is find out what the pre-defined macro is (e.g. sun or sony). A machine-identifier macro such as this is used to identify the machine within the code. You may want to define your own machine-identifier macro within **generic.cfg**.

The next step is to determine how your machine defines values listed in table 3.4.

Many times, these word lengths and such are the same as another machine like the SUN.

3.4 Using `Install`

Some machines require a different makefile or source code structure. For the machines listed in appendix E, a shell has been set-up that will automatically make the necessary changes for you. All you have to do is type `Install` and specify the appropriate definition file. It will automatically determine the type of machine you are own and make the appropriate changes[2]. If it cannot determine the type of machine, then you will have to make the necessary changes yourself. The following sub-sections describe the type of changes that need to be made.

[2]If **Install** wasn't automatically given executable access privileges during the tar extraction, you may need to type `sh Install` to execute the shell or `chmod +x Install` to give it executable access. In addition, you will probably have to use the **chmod** command on the **bin/remsrc** shell as well.

3.4.1 Makefile changes

The following machines require special changes to the makefile structure.

Stardent
HP-UX 9000

Refer to appendix E (Computer Peculiarities) for more information. If your machine is not listed in appendix E, it would be wise to read through the information there anyway just to become familiar with any problems which might exist on your machine.

3.5 Changes to the SOURCE CODE

The following machines require special changes to the SOURCE CODE.

Stardent
HP-UX 9000
POPS/CRAY
IBM AIX RS-6000

Refer to appendix E (Computer Peculiarities) for more information.

Even if your machine is not listed in appendix E, it would be wise to read through the information there to become familiar with any possible problems which might exist. Each machine has its own peculiarities. As a user on a new machine, it is your task to uncover them. However, there are a couple of things to keep in mind to make your task easier.

First, read through appendix E to get an idea of what peculiarities exist among UNIX machines.

Second, add your machine to **src/rlib/liblamps/getdateid.F**. Each machine has a different way of determining the current date and time. You need to check your FORTRAN manual to see how it is done on your machine.

Third, determine whether your machine can sequentially access fixed record-length files. Some fixed record-length files are accessed sequentially because computers like the CONVEX and HP's can handle them. If your machine cannot, you probably need to convert the files to variable record-length format and access them via the structure used by the SUN and Stardent.

Finally, compile the code and try to interpret the error messages. With a knowledge of the different identifier macros, one can identify places within the code where some machines require special coding. Your machine may be one of them. This is done by using the `#if defined` statements. These are the `cpp` statements that are used to put in the machine-dependent statements. By looking at these lines, one can also get an idea where discrepancies have existed in the past.

Chapter 4

CASE SET-UP

For each grid used, a different CASE DIRECTORY is needed to organize the necessary EXECUTABLES. In addition, it might be useful to have different CASE DIRECTORIES even for cases with the same grid in order to differentiate cases in other ways (e.g., time, input data, etc.).

To create (or reorganize an existing) CASE DIRECTORY, the user needs to :

1. run **Setup user** (if not previously carried out)
2. run **Setup case**
3. place all the appropriate DATA FILES in the appropriate directories
4. create the appropriate INPUT FILES

4.1 Running Setup user

Each user must create his/her own LAMPS 90 environment within a directory separate from the MAIN LAMPS 90 DIRECTORY. To do this, type `Setup user` within the MAIN LAMPS 90 DIRECTORY. You will need to enter the name of the directory to be used for carrying out a simulation (i.e., the user's individual LAMPS 90 directory).

The setup process will create the user's individual LAMPS 90 directory (if it is not already there). It will also place within this directory a file called **Setup** and a file called **user.cfg**.

4.2 Running Setup case

To carry out a simulation, one needs to set up a CASE DIRECTORY structure (see section 2.1.3). To do this, type `Setup case` within the user's directory. You will need to enter information regarding the name of the sub-directory to be used for the case (i.e., the case directory) and the grid dimensions.

The setup process will create the case directory (if it is not already there). It will also place within this directory a bunch of files and sub-directories that are used to compile and run the PROGRAMS.

4.3 Getting DATA FILES

4.3.1 Case-independent data

The only case-independent DATA FILES are

1. climatological, monthly-mean sea-surface temperature data
2. terrain DATA VOLUME

These are only of use if the user is going through the whole INITIALIZATION process with observed data.

These files can be obtained from the Numerical Model Development Facility at Drexel University. Both of these files should be completely portable, although the SST data may need to be reformatted (see appendix B). For information on the terrain DATA VOLUME, see the LAMPS 90 Programmer's Guide.

Note: The terrain data can also be generated from the navy terrain DATA FILE via **terrain** (see Programmer's Guide). The navy terrain DATA FILE is also available, but it is usually kept compressed (i.e., **.Z** suffix) in order to save space. Thus, one has to uncompress it after copying it to your machine (i.e., use `uncompress`).

4.3.2 Case-dependent data

If you are only concerned with the DYNAMIC MODEL, you only need one case-dependent file : the INITIALIZATION DATA VOLUME or the MODEL INITIAL DATA FILE (see LAMPS 90 Programmer's Guide). Both of these files depend on the grid, domain, time, etc., so you need to find someone who has run a similar case. Of these two files, it is recommended that the INITIALIZATION DATA VOLUME be used because the DATA VOLUME format, unlike the MODEL INITIAL DATA FILE format, is portable. The MODEL INITIAL DATA FILE stores the data as real numbers and real number definitions can vary from machine to machine (see appendix C).

If you are only concerned with the POST-PROCESSING of the DYNAMIC MODEL output, you only need the MODEL HISTORY DATA VOLUME (see appendix D).

If you are starting from the beginning (i.e., the INITIALIZATION process) then you need a FIRST-GUESS file (usually the nmc gridded data) and some observations (e.g., soundings and daily sea-surface temperatures). These data files should be put in the **dbs/** directory within the *case/* directory. For more information, see chapter 6.

4.4 INPUT FILES

Each PROGRAM requires instructions that are to be read from the standard input (i.e., the screen). Since it is tedious to type in all of the necessary instructions each time a PROGRAM is run, it is standard practice to create an INPUT FILE for each PROGRAM. The INPUT FILE contains all of the necessary instructions for a particular PROGRAM. The user can then re-direct the input to be from the INPUT FILE rather than the screen. This is done rather easily in UNIX via the less-than sign (<). By appending the less-than sign followed by the name of the appropriate INPUT FILE to the executable name, the input will automatically be read from the INPUT FILE. For example,

```
bin/prog < prog.inp
```

will run the program **bin/prog** and read the instructions given in the file **prog.inp**.

4.4.1 Using the sample INPUT FILES

A sample INPUT FILE for each PROGRAM is provided with the LAMPS 90 package. These sample INPUT FILES are stored in the **config/inp/** sub-directory of the main LAMPS 90 directory. It is easier to use these as a basis for the input files rather than starting from scratch.

4.4.2 Running Setup input

Although copying and editing all of the sample INPUT FILES is less cumbersome than creating them all from scratch, it can still be somewhat tedious. To address this, the **Setup** command can be used within the case directory to do the following.

1. copies all of the sample INPUT FILES to a case sub-directory (of your choosing, e.g., **inp/**), and
2. edits them according to parameters you provide to the **Setup** program, such as
   (a) case identifications (e.g., title, time, lat/long) and
   (b) file names (e.g., INITIALIZATION files, MODEL INITIAL DATA FILE, MODEL HISTORY DATA VOLUME).

For the most part, the resultant INPUT FILES should match well with what the user wants to accomplish. However, the user should still make sure that each INPUT FILE has the proper specifications. For example, in **begin.inp** or **analytic.inp**, the user should make sure that vertical levels are defined appropriately. In addition, in **model.inp** or **modeldry.inp**, the user should make sure that the time step and forecast time are specified appropriately. INPUT FILES to the POST-PROCESSING PROGRAMS (see chapter 6) are highly dependent on the each user and the user should edit them before running those PROGRAMS.

Chapter 5

COMPILING AND RUNNING PROGRAMS

This chapter describes how one goes about compiling and running all of the programs that are part of the LAMPS 90 package. See chapter 6 for the different programs that make up LAMPS 90.

5.1 Compiling PROGRAMS

Before one can run one of the LAMPS 90 programs (see section 5.2), one needs to create an executable from all of the relevant source codes. Since it can be quite difficult for the user (especially the novice user) to keep track of all the different source codes required to make just one executable, the LAMPS 90 package includes a makefile system that greatly simplifies the compilation process.

To take advantage of the makefile environment, one must be in the appropriate CASE DIRECTORY. To compile code in order to create the EXECUTABLE (ex. **lyrmean**), type

`make` *exec*

where *exec* is the name of the EXECUTABLE (ex., `make lyrmean`).

Not only does the `make` command automatically know what files to compile to create the executable but the code will only be recompiled if the EXECUTABLE file is not present or if any of the following has been changed since the last compilation.

1. SOURCE CODE (ex. **lyrmean.pre**)
2. relevant libraries (ex. **dviosubs.a**)
3. other relevant files (ex., **generic.cfg**)

The EXECUTABLE will then be automatically placed within the **bin/** sub-directory of the user's CASE DIRECTORY. All OBJECT CODES will be placed within the **obj/** sub-directory of the user's CASE DIRECTORY.

One might ask how the `make` command knows which source codes to compile. This information is stored in a series of files that `make` accesses. When you invoke **Setup** to set-up a case directory, all of the files

necessary to run `make` correctly are copied into your CASE DIRECTORY and its associated sub-directories. One of these files is named `Makefile`. Another lists all of the source codes for the particular executable.

This second makefile is the one that lists the dependencies of the particular PROGRAM. For example, the file that lists the source codes for **lyrmean** is **mks/lyrmean.mk** (i.e., the file **lyrmean.mk** located in the **mks/** sub-directory of your CASE DIRECTORY). In fact, for each executable there is one makefile and this makefile lists all of the files required to create the corresponding executable. The reason why each user gets his/her own copy of all the makefiles is so that each user can change the dependency lists independently. In this way, each user can use his/her own version of the source codes. Everything is controlled by the macro definition lists within the makefiles of the **mks/** sub-directory.

The makefile is basically a job control environment so it is not needed to run LAMPS. However, one would need to compile and archive all of the libraries separately, compile all of the relevant source codes and then link all of the required source codes and libraries. If using **make** is not possible, then one could use the makefiles in the **mks/** sub-directory of your CASE DIRECTORY in order to determine the source codes and libraries required for each program.

Note: Knowledge of `make` is not required to run the code. However, `make` errors tend to be cryptic so a good working knowledge of `make` is advantageous. For general information on `make`, refer to **doc/help/make.help** or the appropriate UNIX manual.

5.1.1 Using your own source code

If you want to change some of the source codes for a particular program, you will need to specify that change within the particular program makefile (e.g., **mks/lyrmean.mk**). Each file is referenced by a series of macros. For example, the macro `$(TRGT_DIR)` stands for the directory that contains the source codes for the particular target. If the source code is coming from a different directory, you must change the appropriate line in the program makefile. See the tutorial (**doc/tutorial**) for an illustration of how this can be accomplished.

Note: If you change the dependency list, the makefile system will not automatically recreate the executable unless the source code has been updated since the last time the executable was created. This is because the makefile system doesn't know that you have changed the dependency list since the last time the executable was created.

If you want to use a PROGRAM that is not part of the LAMPS 90 package, you will need to edit the **Makefile** file. Add your PROGRAM name to the `LIST_FILES` macro and create an additional makefile within the **mks/** sub-directory. Just make sure that your new program has a name that is consistent with whatever changes you make in your makefile.

5.1.2 Using your own libraries

Within the **mks/** sub-directory of the CASE DIRECTORY, there is a file called **prog.mk**. This is the makefile that actually carries out the compilation. It also contains information regarding how each library is to be built. It can be editted much like the individual PROGRAM makefiles.

5.2 Running PROGRAMS

5.2.1 Basic run

`Make` is not required to run the LAMPS 90 codes. In fact, all that is needed is to type the name of the EXECUTABLE file. Since the compilation described in section 5.1 produces an executable in the **bin/** sub-directory of the your CASE DIRECTORY, one can start the program by typing

`bin/`*exec*

where *exec* is the name of the EXECUTABLE (ex., `bin/lyrmean`).

Each program expects instructions from standard input. Since it can be tedious to type in all of the instructions each time one runs a program, a series of INPUT FILES has been created for each program. All one needs to do is edit the appropriate INPUT FILE and have the program read the instructions from that file rather than standard input. This can be done as follows.

`bin/`*exec* $<$ `inp/`*exec*`.inp`

where the less-than sign tells the executable to read the input from the file `inp/`*exec*`.inp`.

A simple way of created default INPUT FILES is to use **Setup**. See the tutorial (**doc/tutorial**) for an illustration of this.

5.2.2 Normal run

In many respects, the procedure outlined above is adequate. However, if one can also use **make** to handle many of the tasks involved with running programs. For example, if the user wants to do any of the following tasks, the makefile structure comes in very handy.

1. run the EXECUTABLES in a certain order
2. run the EXECUTABLES while redirecting the input from the appropriate INPUT FILE and output to the appropriate LOG FILE
3. put any graphics files created in the graphics sub-directory
4. time the process and include this information in the LOG FILES

The only requirement is that an INPUT FILE must be present for the PROGRAM to be executed and this INPUT FILE must be named *exec*.**inp**, where *exec* is the name of the PROGRAM, and it must be in the case sub-directory identified via the IDIR macro within the **Makefile** file. If the user has run **Setup** to set-up the input files, then these requirements are automatically met.

To run the EXECUTABLE code (ex. **lyrmean**), type

`make` *exec*`.log`

where *exec* is the name of the EXECUTABLE (ex. `make lyrmean.log`).

The code will only be run if the LOG FILE is not present or if any of the following has been changed since the last run.

1. EXECUTABLE (e.g. **lyrmean**)
2. INPUT FILE (e.g. **lyrmean.inp**)

5.2.3 Forced run

To force the EXECUTABLE code (ex. **lyrmean**) to run whether or not the dependencies are up-to-date (i.e. EXECUTABLE and INPUT FILE), type

```
make exec.log "FORCE=force"
```

This is useful for those times when you have changed the DATA FILES but the INPUT FILE and EXECUTABLE remain the same.

5.2.4 Using other log, input or graphics sub-directories

To find out which sub-directories the makefile will use to store the input, log and graphics filese, check out the macros named `IDIR`, `LDIR` and `GDIR` in the file **Makefile** (within the user's CASE DIRECTORY). These three macros correspond to the input, log and graphics sub-directories, respectively. To access different sub-directories, just change these macros.

During the setup process for the input files, the user defines the case sub-directories that will be used to store the INPUT FILES. This process automatically resets the `IDIR` macro within the file **Makefile**.

5.2.5 Miscellaneous information

Information regarding dependencies is contained in the case makefile (i.e., **Makefile**). Use `make help` for information regarding the various options.

5.3 Makefile Structure

To accomplish what appears to be a simple task of checking dependencies, an apparently complex set of makefiles are required. Basically, there is one main makefile (i.e., **Makefile**), which utilizes two other makefiles (i.e., **prog.mk** and **libark.mk**) and individual makefiles for each PROGRAM. Each makefile uses the list of macro definitions in the **mks/defaults.mk** file.

> *case*/**Makefile**

- dependent upon the case and machine
- is always referenced by `make` first
- contains all of the macros and is separated into six parts
  1. List of definitions. There are the three sub-directories (i.e., `IDIR`, `LDIR` and `GDIR`) and the rest are stored in the **cfg/standard.dfs** file that is included into **Makefile** via the `include` command.
  2. List of executables that can be made.
  3. List of TARGETS for compiling code and producing the executables.

Procedure	*Description*	*Program Order*
`init`	2-D init	**dvutil begin frstg25 sound lyrmean**
		rhanal isenv2p2 vertran wdadjust
`init381`	2-D init	**dvutil begin frstg381 sound lyrmean**
	w/381 frstguess	**rhanal isenv2p2 vertran wdadjust**
`initgd`	init from different grid	**dvutil begin gdinterp wdadjust**
`init3`	3-D init	**dvutil begin sound fgsa3d masa3d wina3d**
		reha3d
`model`	DYNAMIC MODEL	**merge model**
`modeldry`	simple dry model	**merge modeldry**
`trrn`	terrain data vol	**dvutil terrain**

Table 5.1: List of make short-cuts

4. List of TARGETS for running the executables.
5. Targets for help and information.
6. Short-cut targets.

> **mks/standard.dfs**

- contains the standard macro definitions for the make files
- is included in all the makefiles for the default macro definition list. Some of these macro definitions (i.e., `LAMPS`, `DEF_FILE`, `USER_DIR` and `CASE_DIR` are set during the Setup process. The machine-dependent macro definitions are obtained from the definition file (i.e., `$(LAMPS)/config/def/$(DEF_FILE)`).

> **mks/prog.mk**

- contains the EXECUTABLE creation process
- is used by **Makefile**

> **mks/libark.mk**

- contains the library creation process
- is used by **prog.mk**

> **mks/*exec*.mk**

- contains the dependencies for each EXECUTABLE
- is included in **prog.mk** via the `include` line

5.4 Short-cuts for multiple-PROGRAM procedures

Multiple TARGETS can be used with the same make command. For example,

```
make merge merge.log model.log
```

will compile **merge**, run **merge**, then run **model**. Any number of EXECUTABLES or LOG FILES can be specified. For standard procedures which require multiple EXECUTABLES, the short-cuts listed in table 5.1 are available. have been provided.

To run the PROGRAMS in the specified order, type

```
make procrun
```

For example, `make initrun`.

To compile the PROGRAMS in the specified order, type

`make` *proc*`exe`

For example, `make initexe`.

Note: Adding `"FORCE=force"` to the command line with these short-cuts will force the running of the procedure from the beginning step.

Chapter 6

PROGRAMS

The process of making a forecast can be broken down into three parts.

(1) INITIALIZATION - get a consistent dataset for the dynamic model
(2) DYNAMIC MODEL - use the initialization data to create a forecast
(3) POST-PROCESSING - examine forecast fields (e.g., plots)

To compile and run these programs, see chapter 5. The source codes for all of these programs are stored in the appropriate **src/** sub-directory of the MAIN LAMPS 90 DIRECTORY. These files should remain untouched by the user. If the user needs to make changes to these files, a duplicate should be made and placed under the appropriate **src/** sub-directory of the user's case directory. See chapter 5 for information on using one's own version over the version stored in the MAIN LAMPS 90 DIRECTORY.

For more detailed information regarding individual programs, see the LAMPS 90 Programmer's Guide.

6.1 INITIALIZATION

The INITIALIZATION process produces gridded data with which the DYNAMIC MODEL makes a forecast. At this time, one can either use real data (from an NMC analysis and sounding observations) or analytic data.

6.1.1 Real Data

Initializing the DYNAMIC MODEL with real data is a complicated process. It consists of reading in the available data and then interpolating the data to the model grid, checking for internal consistency. There are a number of different programs that can be run (see figure 6.1). The number and order depends on the process desired.

Each program is contained within one executable. Each program reads from and/or writes to an initialization data volume. A data volume is a file with a packed data format specially created for LAMPS 90. As such, it requires a special set of routines to read from and write to the file (see section D.4). For more information on the initialization data volume, see section D.4.2.

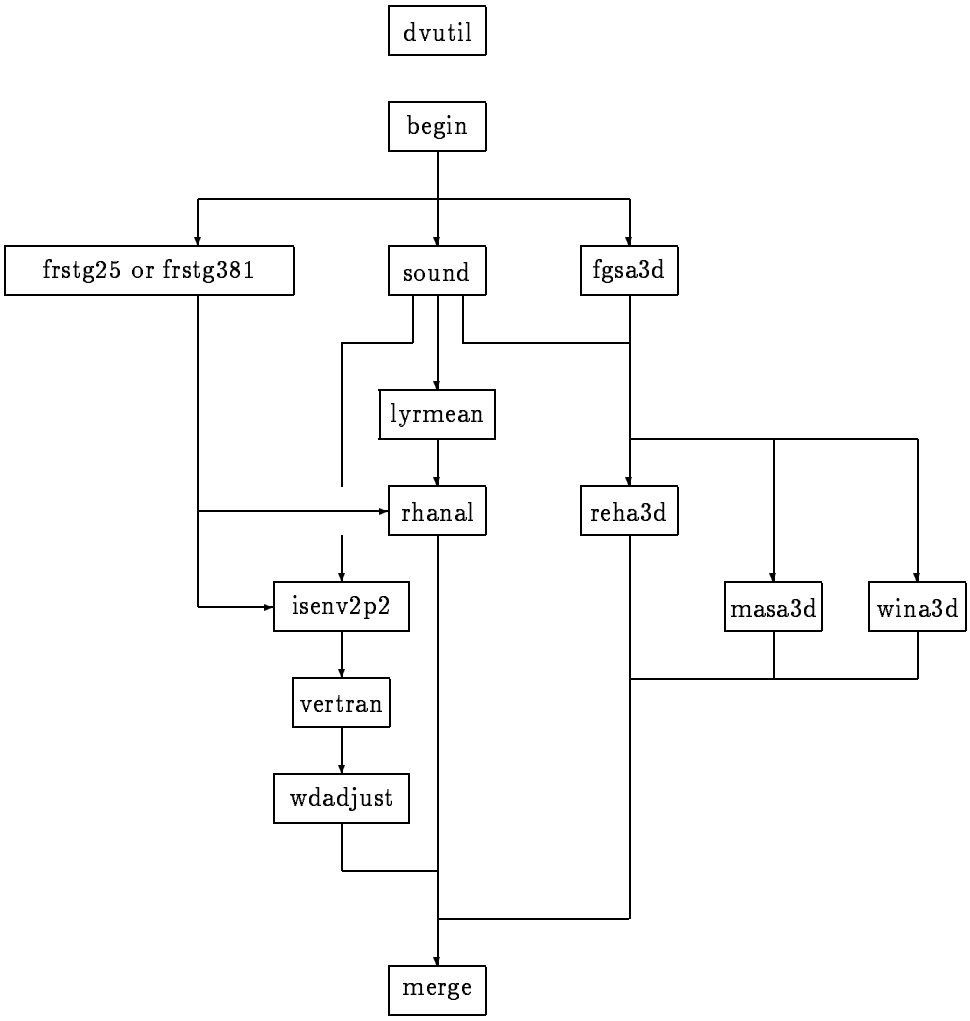

Figure 6.1: Flowchart depicting the sequence of programs needed to create an initial data set with real data. The **dvutil** program is normally used although it is not necessary. All of the programs after **begin** require the output from **begin**. It is also possible to by-pass the **reha3d**, **masa3d** and/or **wina3d** programs and have the **merge** program use the **fgsa3d** output directly.

PROGRAM	FILES NEEDED
dvutil	I/O routines (optional)
begin	terrain data volume SST climatology ERICA 50 km SST (optional) ERICA 14 km SST (optional)
frstg25	NMC 2.5 degree gridded data
frstg381	NMC 381 km gridded data
fgsa3d	NMC 2.5 degree gridded data
sound	sounding files

Table 6.1: Data files needed for the initialization process

Besides the initialization data volume, some of these programs also access other data files. Table 6.1 lists these dependencies.

Because the soundings and NMC gridded data are case-dependent, it is wise to store these data in a sub-directory of the user's CASE DIRECTORY (e.g., **dbs/**). Because the climatological sea-surface temperature and the terrain data are case-independent, it is wise to store these data in a sub-directory of the MAIN LAMPS 90 DIRECTORY or some other common directory.

As referenced to above, the initialization process creates and accesses a data volume file called the initialization data volume. This data volume can be named whatever you want (e.g., `lamps.dvol`). The name is specified within the input files (see section 4.4.2).

Even though the data volume format is portable between machines, the INITIALIZATION DATA VOLUME doesn't contain data which can be read directly by the dynamic model (i.e., the program that actually produces a simulation or forecast). Instead, the **merge** program is used to transform the necessary data from the INITIALIZATION DATA VOLUME to what is called a MODEL INITIAL DATA FILE. It is this file that is used to initialize the DYNAMIC MODEL.

Note: Each step of the INITIALIZATION process may also create a metacode graphics file that is automatically put in the graphics sub-directory of the user's CASE DIRECTORY by the makefile procedure.

6.1.2 Analytic Data

The **analytic** routines provide a simple way of producing data to test the dynamic model without having to go through the complicated initialization process using real data. This program does not create an initialization data volume. Instead, it creates a model initial data file for direct input into the dynamic model (without needing **merge**).

Basically, the **analytic** program uses analytic equations to simulate a certain type of atmospheric structure. See the LAMPS 90 Programmer's Guide for the options available.

6.2 Dynamic Model

A dynamic model is essentially a set of programs that simulate the evolution of the atmosphere through time. There are currently two different dynamic models that can be run within the LAMPS 90 structure.

One uses the full physics (see section 6.2.1) while the other uses a much simpler, dry formulation (see section 6.2.2).

Both programs use the model initial data file to initialize the model variables. This file is not completely portable from machine to machine (see appendix C), so one may want to start from the initialization data volume (i.e., with the **merge** program). If no initialization data volume is available, then either the initialization process needs to be run (see section 6.1) or this data volume needs to be copied from another computer (see section D.4.2).

Note: If time-dependent boundaries will be used, then the dynamic model also needs a time-dependent boundary file (see section D.6).

The dynamic model creates a file that contains a history of the simulation. Like the initialization data volume, it is in the data volume format and so is consequently called the history data volume (see section D.4.3). Since the history is in the data volume format, the history file should be portable from machine to machine. A save file can also be created that contains all the variables from the last output time.

See the LAMPS 90 Programmer's Guide for a list of items that can be specified from within the input file (either **model.inp** or **modeldry.inp**).

All the pertinent routines are in the **src/model** sub-directories (see the LAMPS 90 Programmer's Guide for a list of these files).

6.2.1 Full Physics

The full physics dynamic model consists of the program **model**. This is a full 3-D dynamic model. It includes moisture, radiation, boundary layer and surface energy processes as well as a cumulus/convective parameterization. Options exist for 2-D use, forced 10% relative humidity, zero curvature and noise calculations. In addition, one can use either time-dependent, sponge or cyclic boundary conditions.

6.2.2 Dry Simple

The dry simple dynamic model consists of the program **modeldry**. This dynamic model is structured like the full physics model but it does not include moisture, radiation, boundary layer nor surface energy processes. Options exist for 2-D use, zero curvature and noise calculations. In addition, one can use either time-dependent, sponge or cyclic boundary conditions.

Much of the **modeldry** source codes are very similar to those used by the full physics version. However, each source code file differs in some way. The only exceptions are **flow3dry.F** (**flow3.F**) and **flowmdry.F** (**flowmach.F**). These two files differ only in that the `include` files are different.

6.3 Post-processing

Once the history data volume is created, there are various programs that can access it, reformat it and graph it.

Each of these programs allow for a ".help" option within the input cards. That is, for more information, run the executable code (e.g., **bin/dvutil**) and type `.help`. The corresponding input files also contain descriptions of the possible options available with each program (see the **inp/** sub-directory of the user's CASE DIRECTORY).

6.3.1 Accessing Data Volume Information

The program **dvutil** allows the user to access the data volume information. The following types of access are currently available.

- initialize a new data volume (i.e., set up header record)
- list the contents of a data volume
- add/extract I/O routines to/from a data volume
- copy internal files from one data volume to another
- list the contents of a history data volume header record

6.3.2 Reformatting a History Data Volume

Each internal record of the history data volume, as produced by the dynamic model, contains one "slab" of data for all the variables. A slab contains all of the information from one y-directory grid point (or longitude). That is, each record contains data from an X-Z slab in space of all the variables. This may seem like a strange way to store information but it is used because that is the way the dynamic model stores the information and it is more efficient to write the history data volume with the same format.

Besides appearing a little strange, there are other disadvantages to this formatting scheme. One problem is that to extract an entire 3-D volume for one variable one needs to read through all of the records (since a portion of each variable is stored on each record). Another problem is that the packing scheme works best when the numbers stored on the record are about the same order of magnitude. This means that it is not efficient to store many different types of variables (with different magnitudes) on the same record.

In order to save space and time, the **reformat** program can be used. This program can write out a history data volume that has one variable per record. This format is called the volume format, as opposed to the slab format produced by the dynamic models.

Usually, it is not necessary to rewrite the history data volume. The amount of space and time saved will usually be insignificant unless one is doing a lot of post-processing. However, if you are doing a lot of work with only a few variables, you may want to extract (or derive) only those variables from the history data volume, save them to another history data volume and remove the original history data volume from the disk[1]. This is what the **reformat** program is typically used for. In the process, **reformat** will automatically convert the data to the volume format (if it wasn't already in that format to start with).

For most purposes, the user can remain blissfully unaware of the particular format employed by his/her history data volume. For more information, see section D.4.3.

Note: The **reformat** program can also output data in AVS format for input to the AVS graphics system.

6.3.3 Graphics

The **pltmout** program creates horizontal, cross-section and difference maps from the model history data volume. In addition, it can create vertical profiles including the Skew-T Log-P profile of dry temperature and dew point. It can also create time-series traces of variables. The process uses the **pltmout.inp** input

[1] It might be wise to save the original history data volume on tape before removing it from the disk.

file, which has the specifications regarding the history file name, time to be plotted, variables to be plotted, etc. If the variable being plotted is not in the history file, **pltmout** will try to derive it before giving up.

Note: All graphics produce a metacode graphics file (see section D.8).

Note: **pltmout** can read from either slab or volume-format history data volumes.

Note: To plot the model data within an graphics system external to the LAMPS 90/NCAR graphics system, use the **reformat** program to first reformat the data to the specific format required by the external system. Currently, only output to the AVS (Advanced Visual Systems) field file format is available.

6.3.4 Trajectories

Creating trajectory maps is a two step process.

The first process uses **trajec**, which reads in the data from the history data volume and creates trajectory tracks (see section D.7). Typically, the process uses the **trajec.inp** input file, which has the specifications regarding the data volume file name(s), beginning and ending times of trajectories, starting positions, time step to be used, etc.

The second process uses **trajplt**, which reads in the **trajec** output and creates maps. See the LAMPS 90 Programmer's Guide for information on the type of maps that are currently available.

Note: The file that contains the trajectory tracks, as produced by **trajec**, will be too big if a three dimensional grid of trajectory tracks is requested. Thus, this option is not allowed. Instead, specify that the output is to be stored in the data volume format in much the same manner as a MODEL HISTORY DATA VOLUME file. One must then use **pltmout** instead of **trajplt** to plot the results.

Appendix A

Glossary

CASE DIRECTORY: the directory used to run a case

COMMAND LINES: the instructions for a program, stored in the input file

DATA FILE: a file created or accessed by a LAMPS 90 program.

DATA VOLUME: a LAMPS 90 file which stores data in a format which optimizes space while remaining portable from machine to machine

DATABASE FILE: a file which contains observations

DYNAMIC MODEL: the program which simulates the evolution of the atmosphere through time

EXECUTABLE: a file which contains the information for the computer to run a program - the program will execute automatically when the user types in the name of the executable

FIRST GUESS: a reference state of the atmosphere with which observations can be compared — acts as an initial guess to the real state of the atmosphere

INCLUDE FILE: a file that contains information to be inserted in the SOURCE CODE during the compilation stage.

INITIALIZATION DATA VOLUME: a DATA VOLUME which is created during the INITIALIZATION process

INITIALIZATION PROCESS: the process of taking observed data and creating gridded data for the DYNAMIC MODEL

INPUT CARDS: the same as COMMAND LINES

INPUT FILE: a file which contains commands for a program

LOG FILE: a file produced by a program which contains information regarding the running of the program

MAIN LAMPS 90 DIRECTORY: the directory in which the lamps90.tar was located and from which **linstall** is executed

MODEL INITIAL DATA FILE: the file produced by merge which stores the initial atmospheric state for input to the DYNAMIC MODEL

MODEL HISTORY DATA VOLUME: a DATA VOLUME which is created by the DYNAMIC MODEL and contains a history of the simulation; can be in either slab-format or volume-format (also called history file)

NCAR GRAPHICS: a package created by the National Center for Atmospheric Research to make plots and graphs

OBJECT CODE: SOURCE CODE which has been compiled but has not yet been linked together to form an executable

OBJECT FILES: OBJECT CODE

POST-PROCESSING: code which is used to reformat and graph the information produced by the DYNAMIC MODEL

PROGRAM: a set of routines used for a particular process, which when compiled and linked together with any necessary libraries produce an EXECUTABLE

SOURCE CODE: the listing of instructions in some computer language which tell the computer how to handle the data

SLAB-FORMAT: a type of storage format for atmospheric variables in which each record contains information on every variable but only for one latitude grid point

TARGET: the argument given to the make command

USER DIRECTORY: the personal directory from which each user uses LAMPS 90

VOLUME-FORMAT: a type of storage format for atmospheric variables in which each record contains information on only one variable but for the entire volume

Appendix B

Fixed recordlength, unformatted files

Some FORTRANs (e.g., SUN, Stardent and Cray) cannot handle fixed record-length files unless they are read with direct access. In other words, to read sequentially on some machines requires the use of bookends[1] telling the program how many bytes are part of the next record.

Unfortunately, the style of these bookends vary from machine to machine so none of the data files are stored in this format. Instead, those files that are accessed sequentially are stored in a fixed record length format.

Machines that can access fixed record length files sequentially have no problem with this arrangement. Those that can't, however, need to have the data file transformed into a variable record length file via **bin/fix2var**.

Note: The climatological mean monthly sea-surface data and NMC gridded data are created as fixed record-length, unformatted files. These can be read sequentially Convex and HP's, but not on the SUN, CRAY and Stardent. Before using them, it is necessary to use **bin/fix2var**.

Note: The Sony is unable to handle the `RECORDLENGTH='FIXED'` specification in an `OPEN` statement. The Sun, HP and Stardent just ignore it.

[1] little bits of information surrounding each record

Appendix C

Real number definitions

Real number definitions may vary from machine to machine. For example, reals on the SUN are defined as

$$(-1)^s * 2^{(e-127)} * 1.m$$

while reals on the CONVEX are defined as

$$(-1)^s * 2^{(e-128)} * .1m$$

where s = sign, e = exponent and m = mantissa. As a result, reals on one machine will be interpreted differently by a different machine. In this example, reals on the CONVEX will be interpreted by the SUN as being four times larger than they are supposed to be.

LAMPS 90 uses only one DATA FILE that contains real numbers. This is the MODEL INITIAL DATA FILE. To use this DATA FILE on a different machine, one has to edit the **modelF.cfg** file located within the **cfg/** sub-directory of the user's CASE DIRECTORY. This file contains two cpp macros. One is called `_foreign_`. Defining this macro tells the code to shift the real number definition. The other macro is called `_shftpwr_`. This macro should be defined as the actual power to be shifted.

For example, the Convex defines real numbers differently than the SUN and Stardent. The real numbers on the convex are 4 times larger and need to be multiplied by 2^{-2} (i.e., `_shftpwr_ = -2`) if they are to be used on the SUN and Stardent.

Note: Starting the DYNAMIC MODEL from the INITIALIZATION DATA VOLUME (i.e., with **merge**) avoids this problem.

Note: The terrain DATA VOLUME (i.e., one of the inputs for **terrn04.F**) used to contain real numbers before the DATA VOLUME I/O routines were changed in November, 1991. After this date, the terrain and INITIALIZATION DATA VOLUMES are portable among any machine.

Appendix D

DATA FILES

D.1 Input Cards

Each executable within LAMPS 90 expects instructions from standard input. This information could be entered interactively by the user, but since most of the commands do not change, it is easier to store the commands in an input file and just redirect the input from this input file.

The input files are stored in the input sub-directory of the user's case directory. Each input file is named *inp*/*exec*.**inp**, where *inp* is the name of input sub-directory and *exec* is the name of the executable (e.g., **inp/lyrmean.inp**). It is wise to check these input files to make sure that the information contained within them is what you want (especially for the post-processing graphics).

The format of the input file is very straightforward. Basically, LAMPS 90 obeys the following rules when reading in information from the input files (or the screen).

- all blank lines or lines beginning with # are ignored (in addition, any information after a # is ignored).
- arguments are to be separated by any combination of spaces and tabs.
- principle commands are usually prefixed by a dot (i.e. ".").

Note: The input sub-directory is named via the setep procedure for the input files. Usually, there is only one input directory within a case directory, but multiple input directories may be desired for those cases when multiple runs are being made with the same case, especially if the only thing differentiating these runs are the input files.

Note: For some LAMPS 90 executables (for example, the graphics), one can get a list of acceptable commands by typing `.help`. For others, to learn what options are available, examine the input files configured by the setup process. They may have some documentation explaining the various options possible.

Note: The routines `readcd`, `intarg`, `realarg`, `chararg` and `chararg8` are used to read in information from standard input. See **src/rlib/liblamps/inpctrl.F** for more information.

Navy terrain	Navy terrain information for input to terrain
Monthly mean SST	SST information for input to begin
14/50km ERICA SST	SST information for input to begin
soundings	sounding information for input to sound
first guess	first guess information for input to **frstg25**

Table D.1: Summary of database files

D.2 Database Files

The database files are those files which contain observed data (or data used as a first guess for observed data) — see table D.1.

Depending on the date, these files can be obtained from either NMC or the GALE/ERICA Data Center at Drexel University. All of these files should be portable, although the NMC gridded data (used as the first guess) and the monthly mean SST data may need to be reformatted (see user's guide).

D.3 Log Files

Each executable file sends to standard output any pertinent messages relating to the processing being done. If you are using the makefile procedure outlined in chapter 5, this output is automatically redirected to a log file called *exec*.**log.tmp**, where *exec* is the name of the executable. After successful completion of the program, this log file is renamed to *exec*.**log** and moved to the log sub-directory. It is important that these logs files are checked to insure that the process was completed adequately.

Note: If a process terminates abnormally, the *exec*.**log.tmp** file remains. However, in some instances, depending on how the computer interprets the error, an abnormal termination will appear "normal" to the makefile, causing the log file to be renamed. The user should always check the log files to make sure that the process did, indeed, terminate normally.

Note: The log directory is automatically set to be `log/` but can be changed by editting the `LDIR` macro within the **Makefile** file. Usually, there is only one log directory within a case directory, but multiple log directories may be desired for those cases when multiple runs are being made with the same case.

D.4 Data Volume

In order to save space and ensure portability between any UNIX machine, a packed data file structure was created for LAMPS. This packed file data file structure is referred to as the data volume.

To the FORTRAN programmer, the data volume appears like a directory, with different files embedded within it and with each file containing many records. These are called internal files and internal records. Each internal file is identified by an ID number (called the block number) and a time (usually the initialization time or the history time). Before accessing any data, the FORTRAN programmer must first open the data volume and then specify the internal file to be accessed. This is done via two calls, `opendv` and either `posdv` or `pos2dv`. The programmer can then access each internal record in much the same way as one would access a physical record within a physical file except one uses calls to `readdv` and `writedv` to read and write to the internal record. To skip internal records (forward or backward), one uses the `skiprdv` call.

Multiple calls to `opendv`, with different logical units, opens additional data volumes. If the same logical

Program	explanation	block number
begin	begin a case	1
frstg25	first-guess	2
sound	soundings	3
lyrmean	layer mean	4
rhanal	RH analysis	5
isenv2p2	isentropic analysis	6
vertran	variable transformation	7
wdadjust	wind adjustment	8
anal3d	3-D analysis	9
merge	reformat of variables	10

Table D.2: Block identification numbers in the initialization data volume

unit is used, the data volume previously assigned that logical unit is automatically closed before the new one is opened. The last data volume opened is considered to be "current" and will be accessed by any `readdv`, `writedv` or `skiprdv` calls. To make a different data volume "current" (i.e., to switch the logical unit to be accessed), the programmer needs to use the `switchdv` call.

One can close a data volume oneself via the `closedv` or `close2dv` calls and such a call should always be made before exiting the program in order to close all opened data volumes.

See the *LAMPS 90 Programmer's Guide* for some of the popular data volume calls as well as for information regarding the data volume structure and the packing algorithm.

D.4.1 Terrain Data Volume

To save space, the terrain and surface type information (which is read in by **begin**) is stored in the data volume format. This information is originally stored in a terrain file obtained from the Navy but is very large (about 18 MB). The **terrain** program will extract data for the Northern Hemisphere and store it in the data volume format (about 2 MB).

Note: Because of terrain data volume design, the block ID's do not follow the convention used by the other data volumes (discussed below). Thus, this data can't be merged with the other data volume data.

D.4.2 Initialization Data Volume

Each data record in the initialization data volume includes a block number. The block number identifies the program which produced the data record. Table D.2 lists the various block numbers for the initialization data volume.

It is interesting to note that the block numbers for **frstg25** and **sound** are two and three, respectively, even though two programs usually run in the reverse order (i.e., **sound** before **frstg25**). This is because the order was changed since the numbering was originally assigned.

Note: Currently, **merge** doesn't produce output in the data volume format. Block number 10 is reserved for this use.

D.4.3 History Data Volume

The dynamic model produces a simulation of the atmosphere and saves the state of the simulated atmosphere at specified time intervals. This is known as the "history". The file containing the history is called the history data volume because it uses the data volume format. The data within the history data volume is stored in one of two formats: SLAB-FORMAT or VOLUME-FORMAT. The slab-format is produced by the dynamic model. The volume-format is produced by the **reformat** program.

History Data Volume, Slab-Format

The history data volume, as produced by the dynamic model, is structured as follows.

block #	time	file id	contents
11	0000000000	0	initialization time
			title
			grid size
			lat, long, reference latitude
			height levels, Htop
			ID length, # of 3-D and 2-D variables
			3-D and 2-D variables
11	history time	0	grid size
			lat, long, reference latitude
			slab # 1
			slab # 2
			.
			.
			slab # jmax

History Data Volume, Volume-Format

The history data volume, as produced by the **reformat** program, is structured as follows.

block #	time	file id	contents
12	0000000000	0	initialization time
			title
			grid size
			lat, long, reference latitude
			height levels, Htop
			ID length, # of 3-D and 2-D variables
			3-D and 2-D variables
12	history time	0	3-D variable # 1
			3-D variable # 2
			.
			.
			2-D variable # 1
			2-D variable # 2
			.
			.

D.5 Model Initial Data File

Since the dynamic model expects to read in the initial fields in the slab format, **merge** is run on the initialization data volume to produce a model initial data file. This file is commonly named 'init.dat', but can be named anything.

The model initial data file stores the fields as real numbers, so it may not be portable from machine to machine. See user's guide for more information.

D.6 Time-dependent Boundary File

To run the dynamic model while forcing the time-tendencies at the boundaries to be the same tendencies as that observed, one needs to create a time-dependent boundary file. This file is created from **tbound**, which reads in a series of model initial data files and produces one file which can then be read by the dynamic model.

Note: To use time-dependent boundaries with the dynamic model, change the appropriate cpp macros within **modelF.cfg**. This file should be present within the user's case directory.

D.7 Trajectory Listings

By running **trajec**, a sequence of trajectory tracks can be generated. This information is stored in ASCII format so that it can be easily examined by the user.

D.8 GMETA Files

Programs that create graphics will send the output to a file called **gmeta**. Upon normal termination of the process, the makefile will automatically rename this to *grf*/*exec*.**gmeta**, where *grf* is the name of the graphics sub-directory and *exec* is the name of the executable which produced the `.gmeta` file.

The gmeta file can be plotted via the NCAR graphics commands resident on the user's particular machine.

Note: The graphics directory is automatically set to be `grf/` but can be changed by editting the `GDIR` macro within the **Makefile** file. Usually, there is only one graphics directory within a case directory, but multiple graphics directories may be desired for those cases when multiple runs are being made with the same case.

Appendix E

COMPUTER-SPECIFIC PECULIARITIES

E.1 Sun 4 / Sparcstation 1+

The particular Sun that has been used is located at Drexel University, Atmospheric Sciences Group, Department of Physics and Atmospheric Science.

No changes need to be made to the code in order to run on this machine.

The following list describes some peculiarities of this machine.

1. The C-preprocessor will replace macros on comment lines unless there is a quote there. A quote after `C` on each comment line is used to avoid this problem. One also needs to be careful of any ! comment lines that contain variables due to undergo a macro. Replacing the ! with a !" will solve the problem.
2. If there is a variable called `C` undergoing a macro (ex. cloud water), then the `C` in column 1 of the comment line will be interpreted as a variable. To deal with this, all the comment-identifying C-characters can be changed to either asterisk (*) or exclamation point (!). A better solution is to just change the name of the variable.
3. The compiler can't handle lines longer than 132 characters unless a tab is used at the beginning of the line. A continuation line needs to begin with a "<TAB>1".
4. The SUN can't seem to handle fixed record lengths on sequential access. That is, unformatted files must have bookends. This means that any DATA FILES which have a fixed recordlength and are to be accessed sequentially will need to undergo **fix2var** before they can be used on the SUN (see appendix B).
5. Real numbers on the SUN are defined with a different bias than on the CONVEX. For more information, see appendix C (Real number definitions).

- The macros `sun` and `sparc` are predefined.

Note: The compiler for F77 1.3.1 is known to contain some bugs. One of these is that the compiler can't handle `read/write`'s longer than 8192 bytes or so (2048 integers; this also seems to be the limit for the VAX). To combat this, link the OBJECT FILES with the option `-Bstatic`. Another problem has to

do with I/O. Within the flow subroutines, the compiler can't handle the rewinding of the units. To get around this, the unit would have to be closed and reopened. The macro _SUN_F77_1_3_ was defined once to combat this problem but since this only solved one problem (the DYNAMIC MODEL was still unable to run for some reason), this macro option is no longer available.

E.2 CONVEX C1

The particular Convex C1 that has been used is located at Drexel University, Atmospheric Sciences Group, Department of Physics and Atmospheric Science.

No changes need to be made to the code in order to run on this machine.

The following list describes some peculiarities of this machine.

1. The compiler automatically handles line longer than 72 characters.
2. The Convex C1 has a problem with I/O. The line

```
read (10) (buf(i),i=1,100)
```

   will cause the _read_ command to be called 100 times. To get around this, a routine is created as follows.

```
dimension buf(isize)
read (10) buf
```

3. The compiler is very picky about sending constants to a subroutine which are then changed within the subroutine.

- The macros __unix__ and __convex__ are predefined.

E.3 Stardent

The particular Stardent that has been used is located at the Marshall Space Flight Center, NASA, in Huntsville, Alabama. The name of the computer is 'atmos'.

The following changes need to be made to the code in order to run on this machine.

1. The -O2 compilation can perform some dangerous vectorizations. An example is in STEP. The switching loops (with `atemp`) were separated into two loops to handle this. The -O2 compilation also has problems with `do while` loops. For example, the loop

```
ns=0
do while((.not.havesnd).and.ns.lt.nstn)
     ns=ns+1
     havesnd=.not.spv(usnd(ns,lv))
end do
```

   will need to be rewritten as

```
          do ns = 1, nstn
               havesnd=.not.spv(usnd(ns,lv))
               if (havesnd) goto 800 !  exit loop
          end do
800       continue
```

This compiler problem has never really been fixed. It is hoped that an updated version of the FORTRAN compile will overcome these problems.

2. `fsplit` will name the first file **main** if the first line is a pre-processor line (i.e. it starts with the character '#'). The next `fsplit` would overwrite **main** again, so **libark.mk** needs to be changed such that all the files are split up at once.

3. When using `tar` to extract files from tar files, use `xvof`. The `o` will use the current user's account info.

The following list describes some peculiarities of this machine.

1. The compiler can't handle lines longer than 72 characters unless `-72` is used on the compilation command line.

2. The name of a common block cannot be the same as a routine. (For example, `CONST` or `SURFACE` in the DYNAMIC MODEL).

3. The Stardent doesn't have the command `ranlib`. Use `ar ml`.

4. For some reason, the Stardent can't follow the `VPATH`. Thus, that dependency is no longer present in the makefile structure.

5. The Stardent hasn't been tested to see if it can sequentially access fixed recordlength files. If it cannot, any DATA FILES which have a fixed recordlength and are to be accessed sequentially will need to undergo **fix2var** before they can be used (see appendix B).

- The macros `unix` and `stardent` are predefined.

E.4 MSFC/CRAY

The particular CRAY that has been used is located at the Marshall Space Flight Center, NASA, in Huntsville, Alabama. The current version of the LAMPS code has been compared with code that runs on the NASA/CRAY and therefore is expected to run on the NASA/CRAY, but it has not been tested.

The following list describes some peculiarities of this machine.

1. Filenames must be smaller than 15 characters. This affects the **presplit** mechanism that handles `invoke/block` lines. The old version of **presplit** identified blocks by their block name and this often resulted in filenames longer than 15 characters. The current version of **presplit** identifies blocks by their placement within the routine.

E.5 Sun 3 / SUNOS v4.0 / F77 v1.2

The particular SUN that has been used is located at Drexel University. The name of the computer is 'eagle' (apollo). It has since been upgraded to SUNOS v4.1.1 and F77 v1.4.

The following change needs to be made to the code in order to run on this machine.

1. The compiler can't handle `IMPLICIT NONE` statements. These need to be deleted.

The following list describes some peculiarities of this machine.

1. The compiler can't handle lines longer than 72 characters unless -e is used on the compilation command line. Even then, it can't handle lines longer than 132. To deal with this, the `LNWRAP`[1] `PROGRAM` was created to wrap all lines longer than 72 characters.
2. The compiler can't handle comments at the end of a FORTRAN line (i.e., no '!'-type inline comments). `LNWRAP` strips off all comments.
3. `fsplit` can't split up files that have pre-processor commands. `LNWRAP` removes all *#line* directives and was used after the pre-processor.
4. The compile can't handle dimensioning variables within common blocks. These need to be specified explicitly within `dimension`, `real` or `integer` statements.
5. The fortran compiler cannot handle continuation lines which begin with a tab. Thus, the user needs to make sure that all **.f** files have spaces instead of tabs. (note: all **.F** and **.pre** files will have the tabs expanded to spaces during the `LNWRAP` process).

- The macros `unix` and `sun` are predefined.

E.6 Sony/NEWS

The particular Sony that has been used is located at Drexel University. The name of the computer is 'stormy'.

No changes need to be made to the code in order to run on this machine.

The following list describes some peculiarities of this machine.

1. The compiler can't handle lines longer than 72 characters unless `-extend_source` is used on the compilation command line. Even then, it can't handle lines longer than 132.
2. The default `SHELL` within the makefile system seems to be incorrectly defined as the C-shell. Thus, `SHELL` has to be explicitly set to the Bourne shell.
3. The compiler seems to have some problems with simple coding techniques (ex., concatenation and `var=ftn(var)`). Some recoding may be needed.
4. `fsplit` can't split up files that have pre-processor commands. The `-P` option is used with `CPP`[2] to get around this problem.

[1] `LNWRAP` is a macro defined within the makefiles — **lnwrap.f** was originally used but **wrapln.c** is used now
[2] user-defined within the **.def** definition file — see section 3.2.1

5. The macros `sony` and `sonyrisc` are not predefined when `cpp` is invoked. `cc -P` must be used instead of `cpp`.
6. The record length specifier during open corresponds to the number of 4-byte words, not the number of bytes.
7. The argument to a subroutine must be the same length as defined within the subroutine. Otherwise, the sony will give a 'Bus error'. For example, if a character string of length 45 bytes is passed to a subroutine expecting an integer, the lengths will automatically be off since each integer is 4 bytes long.
8. The Sony doesn't store local variables within a subroutine unless a `SAVE` statement is used. The `-static` flag during compilation will solve this problem.
9. The Sony does not allow intrinsics within the parameter statement.
10. The Sony hasn't been tested to see if it can sequentially access fixed recordlength files. If it cannot, any DATA FILES which have a fixed recordlength and are to be accessed sequentially will need to undergo **fix2var** before they can be used (see appendix B).

- The macros `sony`, `sonyrisc` and `unix` are predefined.

E.7 POPS/CRAY

The particular CRAY that has been used is located at Stennis Space Center in Bay St. Louis, Mississippi (CFT77 v4.0.4.4). It is accessed via a SUN that is used as a front-end (called the Gateway Computer System or GCS). The entire system is called the Primary Oceanographic Prediction System (POPS).

There are actually various SUN's that can be used as a front-end to the CRAY. It is suggested that the user use 'spop'. Other machines can be used, but expect very little capability. For example, on 'pops', the user is unable to use `qsub` and `tar` (use `/usr/5bin/pax -r -f` instead).

Note: Old files are not deleted during the tar extraction process (i.e., it just rewrites over what is already there — if the new file is shorter, remnants of the old file still remain at the end). So, you must delete any old files that are present before using tar. This actually speeds up the extraction process since deletion doesn't require that the backup of the old files be accessed, while overwriting old files does.

From the front-end, all compilation and running is done on the CRAY, named 'lsc' (large-scale computer). To run the PROGRAMS, use the NQS batch processing via

```
qsub -x -lT time_limit -lm size_limitmw -q lsc
cd case_directory
make initexe initrun
^D
```

Other useful NQS batch processing commands are

`qstat` *job#*`.spop@lsc`	get the status of job *job#*
`qdel -k` *job#*`.spop@lsc`	delete job *job#*

See POPS User's Manual for more information.

The following change needs to be made to the code in order to run on this machine.

1. For C routines that are linked to the fortran PROGRAMS (i.e., **src/rlib/gsbytes/gsbytes.c**), the names of the C routines need to be capitalized without the underscore. That is, the user needs to manually remove the underscores and capitalize the routine names.

The following list describes some peculiarities of this machine.

1. The CRAY has a default record-length limit of 267 bytes for formatted reads. If the file is variable recordlength (e.g. **lnwrap** or **wrapln** preprocessing), then this can be gotten around by reading in one byte at a time via an unformatted read (see `rdline` within **lnwrap**) or via a C-language PROGRAM (see **wrapln.c**). If the file is fixed recordlength (e.g. soundings), then this can be gotten around by opening the file as unformatted and reading in a X-length character buffer, where X is the recordlength. The buffer can then be read from in 267 byte or less chunks. Another way to get around this is to use the following compiler options :

   ```
   cf77 -Wl'-D SET=$RBUFLN:4476;COMMONS=$RFDCOM:4485' temp.f
   ```

2. The shell already uses `TARGET` as a macro, so `TRGT` is used instead.
3. `IPATH` is being out-dated. For this reason, it is no longer used within the LAMPS 90 makefile structure.
4. `cpp` interprets the argument after the filename as the output filename. Thus, all flags must be used before the filename.
5. The `Q` format is not allowed in I/O.
6. For some reason, `data tab/x'09'/` is not allowed. To deal with this, all character constants of this type are defined as integers and then compared to the corresponding ASCII character set, e.g., `char(tab)`.
7. List-directed (i.e. default `*`) I/O is not allowed on internal files. I don't know why this is.
8. `ENDFILE` can be used within sequential, unformatted files, but it apparently puts an <EOF> which is not written over with the next write. For those instances when an `ENDFILE` is not followed by a `REWIND` or `CLOSE`, the `ENDFILE` may be unnecessary and should be removed.
9. The CRAY will find duplicate subroutines named `PACK, UNPACK, GETPOS` and `ABORT`. These were not renamed, however, because the LAMPS 90 routines were used automatically. A warning message is produced at compile time, however.
10. The CRAY does not have `lnblnk` defined as a FORTRAN function.
11. All `I`-format I/O must be followed by an integer constant.
12. The CRAY does not allow intrinsics within the parameter statement.
13. The CRAY doesn't store local variables within a subroutine unless a `SAVE` statement is used. The `-a static` compilation flag (i.e., `cft77`) will solve this problem.
14. The CRAY has a problem with doing too many internal make calls. You may have to separate the make calls into separate calls in order to get around this problem. For example, suppose `make a` checks the existence of b which in turn calls `make b` which checks the existence of c which in turn calls `make c`, etc.
15. `cft77` does not allow for multiple files to be compiled at once. The `cf77` command (with `-Wf` options) allows the user to compile more than one file at a time.
16. The Cray hasn't been tested to see if it can sequentially access fixed recordlength files. If it cannot, any DATA FILES which have a fixed recordlength and are to be accessed sequentially will need to undergo the PROGRAM **fix2var** before they can be used (see appendix B).

• The macros `_UNICOS`, `cray` and `unix` are predefined along with some others as specified for CX/CEA systems.

E.8 HP-UX 9000

The LAMPS 90 package has been run on four different HP's. Two were loaners (an HP-720 located at Drexel University and an HP-730 located at NASA's Marshall Space Flight Center). Two HP's are currently being used at Drexel University (an HP-710 and an HP-720).

The following changes need to be made to the code in order to run on this machine.

1. For C routines that are linked to the fortran PROGRAMS (i.e., **src/rlib/gsbytes/gsbytes.c**), the names of the C routines need to be without the underscore. That is, remove the underscores.
2. The HP seems to have a problem with the `EXIT` call. `EXIT` is present but causes a problem with the makefiles (error code 224 - I think `EXIT` is not expecting any parameters). The best way to get around this is to :
   (a) COMPILE **bin/uexit.c** (`cd bin; make uexit.o`)
   (b) ADD `$(BIN_DIR)/uexit.o` to the `F77_LD` line in the **prog.mk** file (as the first OBJECT FILE linked — see **prog.mk**)

   Another option is to replace the `uexit` call within **gpur2.F** with `call abort()`. This, however, also creates an undesirable core dump.

This machine has the following peculiarities.

1. `fsplit` does not allow for multiple versions of the same name. Thus, if the first routine in more than one file is automatically named **MAIN.f** (which will be the case if there are a lot of extraneous lines before the first program statement), then routines will be lost. The **libark.mk** file was edited to change the name of the **MAIN.f** file.
2. Before the 9.0 version FORTRAN release, `fsplit` did not name files past an underscore. Thus, a routine named `open_a` and a routine named `open_b` would both be written to a file named **open.f**. Even though the HP can now handle this, library subroutines should not have underscores just in case other computers have this problem.
3. The loaner HP did not have many extra routines like `idate, exit, lnblnk`, etc unless one added the `+e` flag to the `F77` command line.

   Note: `+e` also accomplishes the same thing as `-K`, which provides for the saving of variables in routines.

   Note: `+e` will add a little bit of time to the run time.
4. The HP does not have `GETARG`, which is used in the `presplit` step. To get around this, a subroutine `getarg` is created that utilizes the `igetarg` function :

   ```
   lenarg = igetarg (iarg, str, lenstr)
   ```
5. Three special things to watch out for in handling scratch files.
   (a) One can't use named scratch files. This will cause an error during run-time. Remove any references to scratch filenames or directories
   (b) To change the directory in which scratch files will be created, use ...

   `setenv TMPDIR` *directory*

   (c) Logical unit 7 is reserved and may cause some problems. Use logical unit 8 instead.
6. Routines which have names similar to common FORTRAN statements (e.g., `WRITE` and `READ`) will cause problems during runtime. Use routine names which are not the same as FORTRAN statements.

7. The HP can sequentially access fixed recordlength files, if they are opened as direct access and then accessed sequentially (with no rec specifier).

- The macros `hpux` and `unix` are predefined along with some others.

Note: The HP doesn't seem to stop during a "divide by zero". Instead, the number is represented as "NaN" (not a number) and all subsequent calculations cause further "NaN"s. To force the execution to stop, use the `+T` flag during compilation.

E.9 IBM AIX RS-6000

One RS-6000 has been tested. It was located in the division of graduates and investigation (DGI) at the following facility in Mexico.

Technological Institute of Higher Education of Monterrey (ITESM)
Mexico State Campus (CEM)

or

Instituto Tecnologico y de Estudios Superiores de Monterrey (ITESM)
Campus Estado de Mexico (CEM)

An in-depth trial was not performed on the RS-6000. Instead, only **analytic**, **modeldry** and an old version of **reformat** were tested. Along the way, it was found that the RS-6000 can't go from **.F** to **.f** automatically. It was not tested in regard to the ability to handle lines longer than 72 characters.

The following changes need to be made to the code in order to run on this machine.

1. For C routines that are linked to the fortran PROGRAMS (i.e., **src/rlib/gsbytes/gsbytes.c**), the names of the C routines need to be without the underscore. That is, remove the underscores.
2. The compiler has a 500 byte limit for character strings. There are places, however, where the character strings are set to lengths larger than 500.
   - INITIALIZATION from observations
     Since the INITIALIZATION from observations was not run on this particular RS-6000, no attempt was made to get around the problem in the INITIALIZATION PROGRAMS.
   - INPUT CARD routines
     The maximum length is currently set at 1024. The compiler will automatically reset this to 500, so this is not a problem (except for the warning messages generated).
   - DATA VOLUME
     The DATA VOLUME has a default record length of 1024 as set by the dvsubs.cfg file. This was reset to 256. However, the user must keep in mind that any DATA VOLUMES created on other machines with record lengths greater than 500 will not be readable on the RS-6000.

   It is expected that there is a f77 flag which can extend this limit, but no f77 documentation was available.

This machine has been found to have following peculiarities.

1. The RS-6000 did not have many extra routines like `idate`, `exit`, and `lnblnk`. It is unknown whether there exists an option to the `f77` compiler to add these routines. The getdateid routine will only set the date/time values to zero.

2. In order to have the macros `_IBMR2` and `_AIX` predefined, the user needs to use `cc -P` instead of `/lib/cpp`. However, `cc -P` automatically renames the output to have a **.i** suffix. Therefore, `/lib/cpp` is used. Since the only macro found to be predefined by `/lib/cpp` was `__EXTENDED__`, this was the macro which was used to identify the RS-6000.
3. All `x` formats must have a field width present. Otherwise, the `x` format will be ignored. That is, all `x` formats should be replaced with `1x`.
4. The `f77` compiler cannot handle continuation lines which begin with a tab. Thus, the user needs to make sure that all **.f** files have spaces instead of tabs. (note: all **.F** and **.pre** files will have the tabs expanded to spaces during the course of wrapln).
5. The `Q` format is not allowed in I/O.
6. The RS-6000 does not have `lnblnk` defined as a FORTRAN function.
7. The RS-6000 hasn't been tested to see if it can sequentially access fixed recordlength files. If it cannot, any DATA FILES which have a fixed recordlength and are to be accessed sequentially will need to undergo the PROGRAM **fix2var** before they can be used (see appendix B).

E.10 Silicon Graphics / IRIX

The particular SGI machine that has been used is located at the Universities Space Research Association in Huntsville, Alabama.

No changes need to be made to the code in order to run on this machine.

The following list describes some peculiarities of this machine.

1. The compiler can't handle lines longer than 132 characters unless a tab is used at the beginning of the line. A continuation line needs to begin with a "<TAB>1".
2. The FORTRAN interprets recordlengths of files to be in number of words (4 bytes/word). The flag `-old_rl` is used to be able to use number of bytes instead.
3. The FORTRAN flag `-static` needs to be used in order to insure static allocation of local variables.
4. A compile bug was evident during the incorrect compilation of a look in **rhanal.pre**. Additional lines were added to work around the compiler bug.
5. An `external iocom` line had to be added to **dvutil.F** and the subroutine `opendv` to insure the inclusion of the block data.
6. Unformatted, direct-access files can be read sequentially.
7. Subroutine `bleck` had to be separated from the **sounding.pre** file in order to avoid potential memory problems with a very large file.

- The macros `sgi` and `unix` are predefined.

Appendix F

PREPROCESSOR LANGUAGE

Although each PROGRAM is designed for FORTRAN 77, certain non-standard FORTRAN directives are needed. Some of these are almost standard in present FORTRAN compilers (like DO-loop structure and DO WHILE's) while others require a separate pre-processor (like the C-preprocessor). Each pre-processor is described in the following sections.

F.1 C-preprocessor (cpp/F)

Since UNIX machines are C-language based, it can be assumed that your UNIX machine already has a standard C-preprocessor. Usually this is called cpp. In some cases (like the SUN), the FORTRAN compiler is configured to handle the C-preprocessor statements automatically. Otherwise, cpp has to be invoked separately.

Only three types of C-preprocessor commands are used.

```
#define

#if/#else/#endif

#include " "
```

These are pretty self-explanatory. The language is C, so within the `#if` statement, use `||` for .OR., use `==` for .EQ., use `&&` for .AND., etc. The `#define` statement can be used to define macros (for global replacement purposes) or to define the machine. For example, some macros are pre-defined on certain machines (but not defined as anything in particular).

> The macros `sun` and `sparc` are predefined on the SUN 4.

> The macros `__unix__` and `__convex__` are predefined on the Convex.

> The macros `unix` and `stardent` are predefined on the Stardent.

> The macros `unix` and `sun` are predefined on the SUN 3.

> The macro `unix` is predefined on the Sony.

To check if a macro is defined, use

```
   #if defined(_convex_)
```
or
```
   #if !defined(_convex_)
```

F.2 presplit-preprocessor (pre)

A feature that is commonly used in PROGRAMS written with IFMAC is the `INVOKE/BLOCK` structure. This is similar to the `INVOKE/PROCEDURE` structure used within PREP7 on the Convex. It is also similar to the internal subroutines that are expected to be present within the FORTRAN 9x compilers.

Since the current FORTRAN compilers cannot handle this structure, **presplit** was created to separate the blocks from the code and include them wherever they are needed. No changes need to be done to IFMAC structure of blocks in order to run **presplit** unless the block name utilizes a continuation statement.

F.3 lnwrap-preprocessor

With the use of macro replacements, many lines can grow to lengths longer than 72 characters (and perhaps past 1000 characters). Most computers have a way to deal with long lines (at least until 1000 characters). You just have to figure out what it is. Some machines, however, are not designed to handle such long records. To deal with these machines, **lnwrap** and **wrapln** were designed. They will automatically wrap all lines greater than 72 characters in length. In addition, they will remove all the comments at the end of a FORTRAN line (usually delineated with a `!` or `;` character) and remove all #-line directives.

lnwrap was originally written in FORTRAN but encountered problems on machines that did not allow long record lengths to be read from formatted files (ex. CRAY maximum is 267 bytes). Changing the read process to unformatted read, one byte at a time slowed down the process 20 fold. Thus a C-language PROGRAM (**wrapln.c**), written by X. J. Hou at Drexel University, is now used to deal with the line-wrapping.

F.4 ifmac-preprocessor (im)

The above non-standard language structures have traditionally been handled by the pre-processor IFMAC. IFMAC contains FORTRAN-type commands (IFTRAN) as well as C-preprocessor-type commands.

F.4.1 IFTRAN / FORTRAN differences

Some of the differences between IFTRAN and FORTRAN are listed here.

- `IF` statements do not have an accompanying `THEN` (although standard FORTRAN can also be handled through IFMAC).
- `OR IF` is used rather than `ELSE IF` (although standard FORTRAN can also be handled).
- `DO` loops incorporate parentheses. For example, `DO (I=1,10,1)`
- `FOR` loops can also be used. For example, `FOR (I=1 TO 10 BY 1)`

- Other variations on DO-type loops are as follows.

```
FOR (I=1 TO 10 WHILE (.NOT.END) )
FOR (I=1 TO 10 UNTIL (END) )
LOOP/EXIT/ENDLOOP
```

F.4.2 IFMAC / CPP differences

Although there are many IFMAC commands, the structure of the IFMAC macro- capability is similar to the C-preprocessor except for the language. For example,

IFMAC command	C command
.SET	#define
.IF/.ENDIF	#if/#endif
.USE	#include

F.5 Converting from IFMAC to pre

Many PROGRAMS for LAMPS were written in IFMAC, so it is possible that one might encounter an old LAMPS PROGRAM written in IFMAC. Most of these PROGRAMS were converted to pre for use in LAMPS 90. However, if this is not the case and the user needs to use the PROGRAM, it is possible for the user to convert the code. Converting to **.pre** (i.e. to be pre-processed by **presplit** and cpp) may seem daunting but is relatively straightforward.

- Use **ifmac2pre** to convert the **.im** files to **.pre** files.

  Note: This will automatically convert the file from upper-case to lower-case. As such, you may be required to change some things manually. First, the INPUT FILE, CONFIGURATION FILE and any included files (and filenames) also need to be converted to lower-case. Second, some NCAR GRAPHICS routines may require upper-case arguments (i.e. PWRITX, MAPSTI, MAPSTC, MAPROJ, MAPSET, SETUSV, etc.).

- Change all UNTIL's, WHILE's, REPEAT's and LOOP's into DO-loop or DO-WHILE construction.

  Note: These type of loops can be handled with GOTO's. If you would rather use DO WHILE's, one way of converting to FORTRAN is as follows.

```
DO (I = A, B WHILE .EXP.)          I = A
          .                        DO WHILE (I.LE.B .AND. .EXP.)
          .                             .
          .                          I = I + 1
END DO                             END DO

DO (I = A, B UNTIL .EXP.)          I = A
          .                        DO WHILE (I.LE.B .AND.
          .                           (I.EQ.A.OR..NOT..EXP.))
          .                             .
          .                          IF (.NOT..EXP.) I = I + 1
END DO                             END DO
```

  Note: The LOOP/EXIT structure can be tricky. It may be best to use GOTO's until FORTRAN 9x comes out at which time the LOOP/EXIT structure can be handled.

- Change the CONFIGURATION FILE.

  This may be messy. Basically, extract all the `.SAVE` portions into individual files (use the same naming convention but add a `.h` to the filename). Then change all the `.SET` lines to `cpp` language. Any CONFIGURATION FILES must be included via the `#include` command at the beginning of the PROGRAM.

  Note: Many of the definitions in the particular configuration file may already be set within the two INCLUDE FILES of LAMPS 90, **generic.cfg** and **case.cfg**.

Note: Since FORTRAN 9x is expected to allow blocks (internal procedures or subroutines), it may be best to keep the block structure for now (and use presplit). However, if you are planning to convert blocks to subroutines, the following technique has been used. First use `IMPLICIT NONE` and pass all the variables via a common block (include a **common.h**). Then, step by step, individual variables can be deleted or passed as necessary.

F.6 Converting from pre to ifmac

Much of the LAMPS 90 code is already configured for IFMAC. Use **pre2ifmac** to convert from pre to ifmac. This will do most of the transformation for you. However, you will still need to organize the files appropriately and convert the CONFIGURATION FILE.

Note: **pre2ifmac** will automatically convert the code to upper-case and remove all tabs.

Appendix G

PROGRAMMING PHILOSOPHY

These chapter describes the philosophy governing the structure of the SOURCE CODE, DATA FILES and directories. This is especially useful for those users who are planning to examine or make improvements in the code. It is felt that it is very important that the programming within LAMPS 90 follow one particular philosophy. This makes it easier for users to follow all sections of the code. The disadvantage is that the programmer must keep the philosophy in mind when creating new code.

G.1 Files

The file structure is pretty straightforward. Although each PROGRAM may require many different routines, those which are only used for that PROGRAM are found under the appropriate src/ sub-directory designated for that PROGRAM. These should be grouped into a file structure that makes sense. That is, although one file per routine is convenient for programming purposes, routines should be grouped into files with each file containing those routines that have some identifiable connection.

Routines used by many different PROGRAMS (e.g., routines for DATA VOLUME I/O, graphics and mathematics) should be stored separately within the **src/rlib/** directory.

G.2 PROGRAMS

G.2.1 SOURCE CODE

- Each PROGRAM should contain a main PROGRAM that serves as a guide to the various steps that will be performed in carrying out the particular model process. The main PROGRAM doesn't deal with the details of the model process. Instead, it should call various routines which then deal with the details.
- Run-time options should be input using the `readcd` process. A `readcd` call reads a line of input (from standard input) and returns the number of arguments present. Using `intarg`, `realarg`, `chararg` and `chararg8` returns the value of the specified argument.
- Each identifiably-separate step should be "housed" in either a procedure (i.e. invoke/block) or subroutine. This is one way to force "self-documenting" code.
- Lower-case is preferred.

- A standard FORTRAN indent should be followed. This allows for easy reading by the user.
- The redundant use of `return` or `stop` before `end` should be avoided.
- The use of `call exitx` rather than `stop` is recommended. In this way, the status of the exit can be utilized.
- The use of subroutine calls for printing (like `print` and `string`, which write out various strings) can be used if the common block /iountc/ is not specified. However, if it is, it is probably best to avoid these subroutine calls because they can be cryptic to the user.
- For error manipulation and exiting, use `abend` and `egress`. While this can be cryptic, it is easier to handle computer peculiarities and I/O this way.
- The use of empty C-comment lines should be avoided.
- C-preprocessor macro names should be encased by underscores (e.g., `_test_`). This allows for easy identification of macro names.
- The C-preprocessor `#if defined` should be used to insert changes within the code. In this way, changes can be identified and easily included or ignored.
- Care should be taken to create code that runs on any UNIX machine. Some guidance is given in the LAMPS 90 User's Guide. In addition, one should be aware of the use of non-default precision, like the use of `sngl(a)`, where `a` is `real*8`. This will cause an error on the CRAY since a `real*8` variable is not double precision.

Each SOURCE CODE file can be written in standard fortran (**.f** extension), fortran with C-preprocessor (**.F** extension), fortran with block usage (**.pre** extension) or C (**.c** extension). SOURCE CODE written in IFMAC (**.im** extension) can also be used but the IFMAC preprocessor is only available on some machines (CONVEX C1 and MSFC CRAY).

G.2.2 INPUT FILES

INPUT FILES contain the INPUT CARDS that are read by each PROGRAM during run-time. Thus, values that are commonly changed (like the date and process options) should be read from the INPUT FILES. The routines `realarg`, `intarg`, `chararg8` and `chararg` (within **inpctrl.F**) allow for relatively easy I/O from the INPUT FILES.

Note: These are somewhat equivalent to the `CARD`, `IFIX(CARD)` and `KARD` functions used with the old LAMPS.

Each INPUT FILE should have an **.inp** extension.

G.2.3 INCLUDE FILES

In certain cases, several different routines need to access the same information. This information is provided to the routine by inserting the same information in each routine. The file that contains this common information is called an INCLUDE FILE because it is only inserted during the compilation stage. In the actual source code, there is a pre-processor statement that specifies that the information be included during compilation. For example,

```
#include "file.h"
```

specifies that the file **file.h** be inserted at that place in the code.

The information in the INCLUDE FILE can be expressed in one of two ways. It can be a bunch of C-preprocessor macro definitions, in which case the file must be inserted at the top of the source code file (the definitions only apply to everything following it — by inserting it at the top of the source code file, the definitions will apply to everything in the file). By convention, this type of file will have a **.cfg** suffix.

The other way to provide the information is to express it in a bunch of FORTRAN parameter statements or common blocks. This type of file must be included at the appropriate place in each applicable routine. By convention, this file will have a **.h** suffix.

In many ways, a .cfg file and a .h file are the same. However, the .h files can also be used to store common blocks. With the use of many subroutines, it may be necessary for many subroutines to access the same, long list of variables. Rather than passing them through an argument list, a common block or parameter list is frequently used. In order to avoid the need to change every occurrence of a particular common block whenever a problem arises, it is recommended that one stores these common blocks in a separate INCLUDE FILE which is then included into the SOURCE CODE at the appropriate place via the `#include` line.

Note: Watch for problems with `common`'s. Some computers may be sensitive to fancy programming with `common`'s. For example, it is a good idea to separate character strings from real/integers.

G.3 Libraries

Routines for graphics and mathematics should be stored in sub-directories of the **src/rlib/** directory for use by all PROGRAMS. Refer to the LAMPS 90 Programmer's Guide for information on the libraries. It is strongly suggested that these standard mathematical and meteorology routines be used if possible in order to avoid problems with redundancy.

Note: Routines for the NCAR GRAPHICS library are usually stored in a separate directory that deals with the particular graphics package.

G.4 Names of common variables

Confusion can occur when various routines use different variable names to refer to the same variable. It is recommended that the following guidelines be followed.

- use `luinp` (or `icrd`) and `lulog` (or `ilpr`) for the logical units to standard input and output.
- for constants such as the acceleration of gravity, the universal gas constant and the molecular weight of dry air, use the constants as defined in the **incl/const.h** file (or via the `const` subroutine in **src/rlib/liblamps/gpur2.F**).