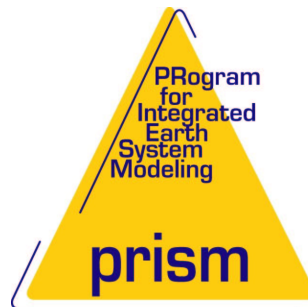


PRISM
Project for Integrated Earth System Modelling
An Infrastructure Project for Climate Research in Europe
funded by the European Commission
under Contract EVR1-CT2001-40012



The PRISM
Standard Compile Environment
Handbook

Edited by:
Stephanie Legutke and Veronika Gayler

PRISM-Report Series-04

0. Edition (release prism_2-4)
(last change: January 19, 2005)

Copyright Notice

© Copyright 2003 by PRISM

All rights reserved.

No parts of this document should be either reproduced or commercially used without prior agreement by PRISM representatives.

How to get assistance?

Assistance can be obtained as listed below.

PRISM documentations can be downloaded from the WWW PRISM web site under the URL:

<<http://prism.enes.org>>

Phone Numbers and Electronic Mail Adresses

Name	Phone	Affiliation	e-mail
Stephanie Legutke	+49-04-41173-104	MPI-HH, M&D	legutke@dkrz.de

Contents

About this handbook	1
Introduction	1
1 Source code management	3
1.1 The PRISM SCE directory structure	3
1.1.1 The <i>root/src</i> directory	4
1.1.2 The <i>root/arch</i> directory	7
1.1.3 The <i>root/util</i> directory	9
1.2 Coding conventions: models and libraries	9
1.3 Coding conventions: scripts	11
1.4 The PRISM CVS repository	12
2 Compiling	14
2.1 Model compilation	15
2.1.1 Low level model makefiles	17
2.1.2 Generating low level model makefiles	20
2.1.3 Component model compile scripts	21
2.2 Library compilation	28
2.2.1 Low level library makefiles	28
2.2.2 The library compile script	29
3 Extending the SCE	32
3.1 Extending the SCE to accommodate a new model	32
3.2 Extending the SCE to accommodate a new site	34
4 Using the GUI	36
4.1 Configuring compile scripts with PrePare	36
4.2 Monitoring the compilation with SMS	36
A Component model compile script generation tool	40
B Component model compile script	45
C Example library makefile	51
D Library compile script generation tool	52
E The library compile script	55

List of Figures

1.1	PRISM SCE directory structure	4
1.2	PRISM SCE source code directory structure	6
1.3	PRISM SCE architecture directory structure	8
2.1	Compile script flow for a main model with submodel	15
2.2	Error message for a missing submodel compile script	16
2.3	High level compile script design	16
2.4	First part of a low level component model makefile	19
2.5	Help text of <code>Append_dependencies</code>	20
2.6	Message printed by <code>Append_dependencies</code> for invalid model name	20
2.7	Messages printed by <code>Append_dependencies</code> for an invalid directory	21
2.8	Messages printed by <code>Append_dependencies</code> for missing <code>Makefile_1</code>	21
2.9	Commands for model compilation	22
2.10	Control messages from compile script creation	23
2.11	Help text for component model compile script creation	23
2.12	Help text of compile scripts	24
2.13	Control messages for submodel compilation	24
2.14	Compile script assembly	25
2.15	First part of the <code>PSMILe</code> low level makefile	28
2.16	Help text for the library compile script creation	29
2.17	Help text for the library compile script.	29
3.1	Example for a model source code storage adaptation to the SCE	33

About this handbook

The PRISM infrastructure software provides support for the full suite of steps that is needed to perform an experiment with a PRISM earth system models. The suite starts with the source code retrieval and ends with the visualization of model diagnostic output. This handbook describes only one aspect of the full system, the PRISM SCE (Standard Compile Environment) covering the first part of the suite: retrieval of the necessary source code and tool box from the repository, generation of Makefiles and compile scripts, and compilation. It is complemented by the handbook on the SRE (Standard Running Environment) by Gayler and Legutke (2004) which describes the next steps of the PRISM experiment suite, the configuration and creation of run scripts and the model execution.

This edition describes the software release tagged as indicated on the cover page. The PRISM software development is an ongoing process as is the writing of this handbook. These activities will be synchronized as much as possible. Therefore, new editions of the handbook will be provided in about the same intervals as new PRISM software is released with major modifications of the SCE. If model adaptation activities are started, it should thus be ensured that the newest edition of the handbook is used. It can be downloaded from the PRISM web site (<http://prism.enes.org>) or, together with the PRISM software, from the PRISM source code repository.

Acknowledgment

The design of the SCE profited much from discussions with colleagues and their input: Marie-Estelle Demory (IPSL, Paris), Luis Kornblueh (MPI-M, Hamburg), Jean Latour (FSE/Fujitsu, Toulouse), Thomas Schoenemeyer (NEC, Munich), Sophie Valcke (Cerfacs, Toulouse), and Reiner Vogelsang (SGI, Juelich).

Introduction

The PRISM project aims at the establishment of a climate research network in Europe. An important step towards this goal is the development of an infrastructure including a flexible, easy-to-use, and portable software for earth system modeling.

Keeping in mind the large number of models and platforms used in Europe for climate modeling, and taking into consideration the quick development of both software and hardware, it is obvious that the PRISM software must be extendable to accommodate new models and platforms, and must facilitate the replacement of component models, while still being low in maintenance.

In order to meet the requirements of portability, flexibility and extendibility at low maintenance costs, the PRISM software is highly modularized and gives the user a common look&feel for all activities in the system.

The PRISM earth system models, on the other hand, have to meet a minimum of standards in order to allow the use of the PRISM tools for model development and automatic processing.

Some of the standards are summarized in the SCE described in this handbook. The specifications of the SCE have been developed with the idea in mind to keep the required changes to the models at a minimum in order that a maximum of models can be included into the system with little effort.

One aspect of the SCE is a standard directory structure in which the model and library source codes are stored. This makes it possible to have a portable toolbox for model compilation that can be used for all models. The modularity of the design minimizes the maintenance costs when the PRISM system develops or new models or platforms enter the system. The source code management is described in Chapter 1.

The compilation is based on the GNU make utility in order that only a minimum of recompilation is done during model development activities. The design of the compilation system, the Makefiles and compile scripts and their generation with the toolbox is described in Chapter 2.

Chapter 3 summarizes the steps which have to be done in order to adapt a new component or coupled model to the PRISM SCE or to include a new platform into the system.

A PRISM experiment can be set up and executed on the scripting level, i.e. the user generates and uses UNIX shell scripts, or it can be configured and monitored through a GUI. This handbook concentrates on the description of the system of shell scripts and tools. The interfaces to the GUI are shortly described in Chapter 4. For details about this topic, however, the reader has to refer to the 'User Interface Guide' of Constanza et al. (2004).

Chapter 1

Source code management

One aspect of the PRISM SCE is a standard directory structure for the storage of model and library source codes, locally and in the central repository. Once a model's source code storage is organized as in the SCE, the tools provided to generate model and platform specific compile scripts can be used (see Chapter 2). The scripts are assembled from a base of fragments of script code. The fragmentation is done in a way that classes of fragments are created which are model specific or platform specific or both, and a class which can be used for all models on all platforms. This has a number of advantages. It is obvious which fragments have to be added for new platforms or for new models, and the same fragments can be used for the Unix shell scripts as for the GUI system. Redundant code is avoided and the maintenance costs of the system are kept at a minimum. In addition, the user gets a common look&feel with all models and platforms.

The first section of this chapter describes the software directory structure with its three main branches, the branch containing the model and library source code (Section 1.1.1) and the branch containing the tools needed for the scripting system (Section 1.1.3). A third branch, described in Section 1.1.2, contains only platform dependent code as e.g. compiler output. It is created when the models are compiled.

Earth system component models may be enabled to run in different coupled constellations. Since only one version of a model source code is maintained, the option for each constellation must be permanently kept in the code. The same holds for the options to run on different platforms. Section 1.2 describes the conventions how to handle this.

Coding conventions exist also for PRISM scripts for both systems, the UNIX shell scripts, and the scripts used with the GUI method. They are shortly summarized in Section 1.3.

The central PRISM repository is archived and administered with the CVS (Concurrent Versioning System) tools. The access to the repository is subject of the last section 1.4 of this chapter.

1.1 The PRISM SCE directory structure

The PRISM standard directory tree on the compile server after download from the CVS repository and after at least one compilation action is displayed in Figure 1.1. The root directory on the CVS repository is `./prism`. If not present locally, this directory is generated when the download is done using the CVS modules defined for the PRISM system. This, together with the name of the working directory from which the download was triggered is called *root* in the following. The *root* directory path name is detected automatically by the PRISM tools and can therefore be changed by the user. All directories below are addressed as downloaded or created by the scripts and the structure as well as the location of files must be kept for the system to work.

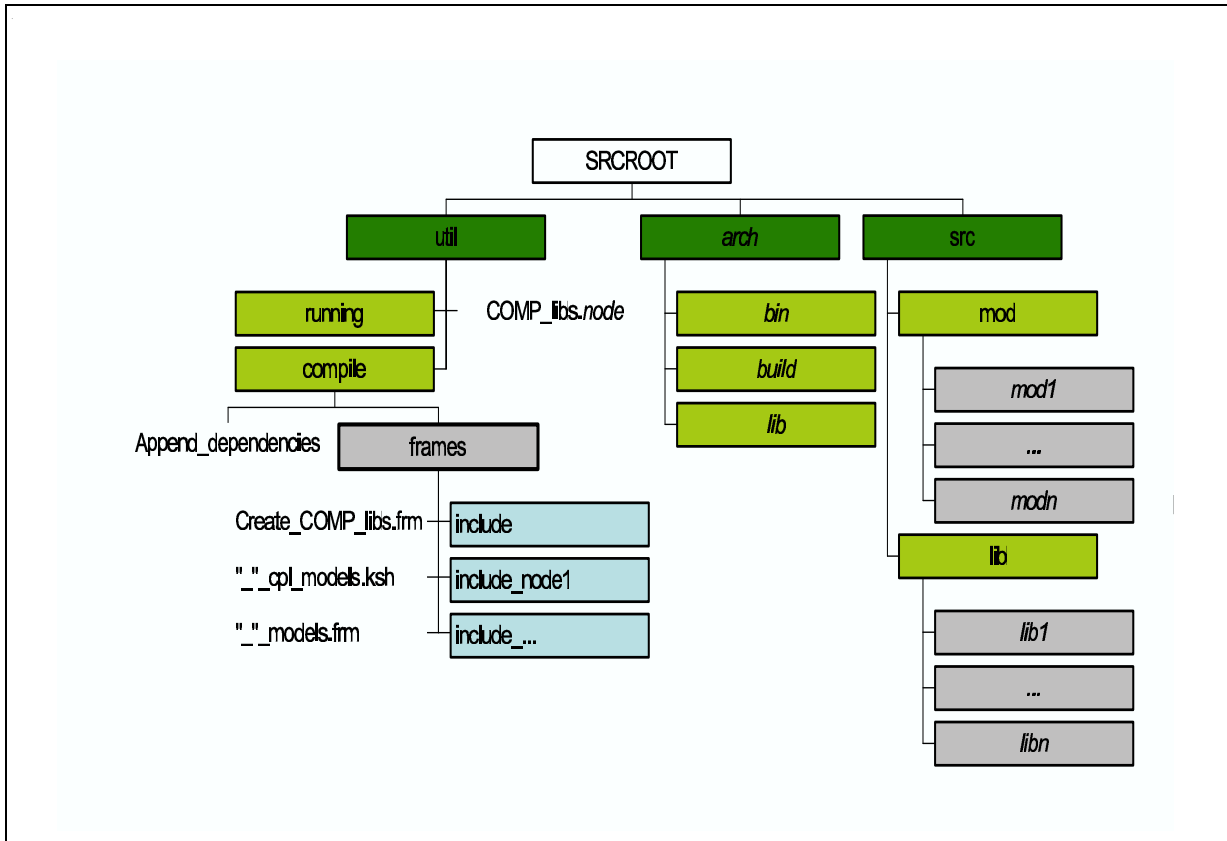


Figure 1.1: The PRISM SCE directory structure. The library compilation script in directory `/root/util/` is not downloaded from the PRISM source code repository. It has to be created for the target platform by the user.

All files and paths are given relative to the `root` directory. From the `root` directory, three major branches separate: `util`, `src`, and `arch`¹.

The first two directories are downloaded from the PRISM CVS repository. The third, the `arch` directory is created during the compilation process by the system. It is called the 'build' directory (see Section 1.1.2).

1.1.1 The `root/src` directory

The `src` directory contains the source code of the component models and of the libraries in the subdirectories `mod` and `lib` respectively.

`root/src/mod`

The `mod` subdirectory contains the source code of PRISM component models. There is exactly one subdirectory for each individual component model. All model directories are named, in lower case letters, after the model it contains (`mod/model`). The name should not contain any release, version, or revision numbers.²

Subroutines, modules, or header files of a model are stored in subdirectories below the `src/mod/model` directory. There can be an arbitrary number of source code directories, and these may contain a mix of

¹All file, variable and command names are written in typewriter font. Variable parts of names are in italic.

²It may be necessary to keep more than one version of a model at the same time at a single site and tag it with the release number or CVS-tag or the like to differentiate between the versions. Be aware that some tools of the PRISM SCE check a given model name against a list of valid models and will not work with any invalid specifications. These tools and workarounds will be noted in the following.

the above mentioned kinds of source code files. It is, however, good practice to keep files, included into other files by the FORTRAN or any other preprocessor before compilation, in a directory named `model/include`. If these are header files they should have an `.h` as a suffix. This `include` directory has to be kept at the same hierarchical level as the other model source code directories. If there is more than one source code directory each of them may have an `include` subdirectory. These subdirectories should, however, only contain code which will be included into files of the parent directory. It is bad practice to have header files with the same name or content in different directories.

The motivation behind these rules is to allow the use of tools to automatically generate scripts for compilation and other purposes. A simple structure for these tools and the scripts they generate is possible only when certain assumptions on the directory tree structure can be made. In the present release the prerequisites for Makefiles are generated by such a tool. It loops over all direct child directories of the model directory, but does not enter another level in the tree. Similarly the compile script loops over all child directories only.

Note that it is not necessary to keep FORTRAN90 modules and subroutines in separate directories. The tool which generates the Makefile prerequisites loops over all directories and searches for prerequisites in all other directories of the same level. Therefore the order of source code directories in the compilation process does not matter.

All directories which contain source code files that are input to a compiler command making a target contain a file named `Makefile_1`. This file is the first part of the Makefile for the `(g)make` command. Presently, the file is written by the model developer. It may not contain any non-portable code. Instead, all platform dependent code of the Makefile is set in the compilation scripts and enters the Makefile via exported environment variables. Help for the creation of `Makefile_1` is given by the tool that generates the prerequisites to a Makefile compilation target (`root/util/compile/AppendDependencies`). It outputs a protocol of all directories containing source code of prerequisites for inclusion into the `(g)make VPATH` variable. Directories containing header files are also detected and output for inclusion into the `(g)make VPATH` variable as well as into the `(g)make INCLUDES` variable specifying the include directories for the preprocessor (see Chapter 2).

The compilation process loops over all directories containing source code other than header files that shall be compiled. The `(g)make` command detects this code by selecting all files with any of the suffixes `.c`, `.f`, `.F`, `.f90`, or `.F90`. All other files will be ignored. Each of these classes has its own rule for compilation. There is no need therefore to have extra directories for files with different suffixes (e.g. `.c` and `.F90`). A larger granularity for compilation rules is not yet supported.³

All model specific documentation which shall be downloaded from the PRISM CVS repository should be stored in a directory named `doc`. This directory should not contain input to the compilers since it will be ignored by some tools.

Main models, submodels, and libraries

If required there may be more than one source code directory in a model directory in order to group routines of related functionalities (e.g. all cloud physics related routines). However, this does not apply to physics or parameterizations which are essentially equivalent to PRISM defined component models (e.g. sea ice or land surface schemes). This code, if originally contained in a source code directory of the calling model, should be moved into a separate component model source code branch (`src/mod/submodel`). If a parameterization as e.g. cloud physics is intended to be used by more than one model, it should also be moved into its own `src/mod/submodel` directory. This allows for a design of the SCE supporting a flexible exchange of submodels or parameterizations in component models.

The question whether a set of source code files should be contained in a `src/lib` or `src/mod` sub-

³It is recommended in the case that some files need different compiler options to gather these files into a new directory. A differentiation of compiler options is then easily done.

directory is decided according to whether the files contain other than platform dependent conditionally compiled code (wrapped with `cpp` flags). The same compiled library is used by all models on the same platform. It may therefore contain only platform dependent conditional source code.

In contrast, model source code may depend also on specifications such as the grid or the other models it will be coupled to. Therefore, the compiled source code of models will be tagged in order to differentiate between differently configured versions of the same model on the same platform.

No matter whether a set of source code is a library or a submodel called by the main model, it must meet the 'package rule' which requires that a parameterization or submodel may not use any information from the calling model (see Mangili et al. (2003)) other than what is passed in the parameter lists of the subroutine calls. Included files (header files and modules) must necessarily reside in the model directory branch where it is used in order that the user can rely on the (`g`)make feature to only trigger a minimum of compilation. The tool detecting the search directories does not leave a model's directory tree to search for unresolved prerequisites. An exception is the common use of modules that guarantee the use of the same FORTRAN KIND specifications of these parameters as it is done with the MPI library include files `mpi.h` or `mpif.h`. Similar include files exist in the PSMILe (PRISM System Model Interface Library) library for the coupling exchange data.

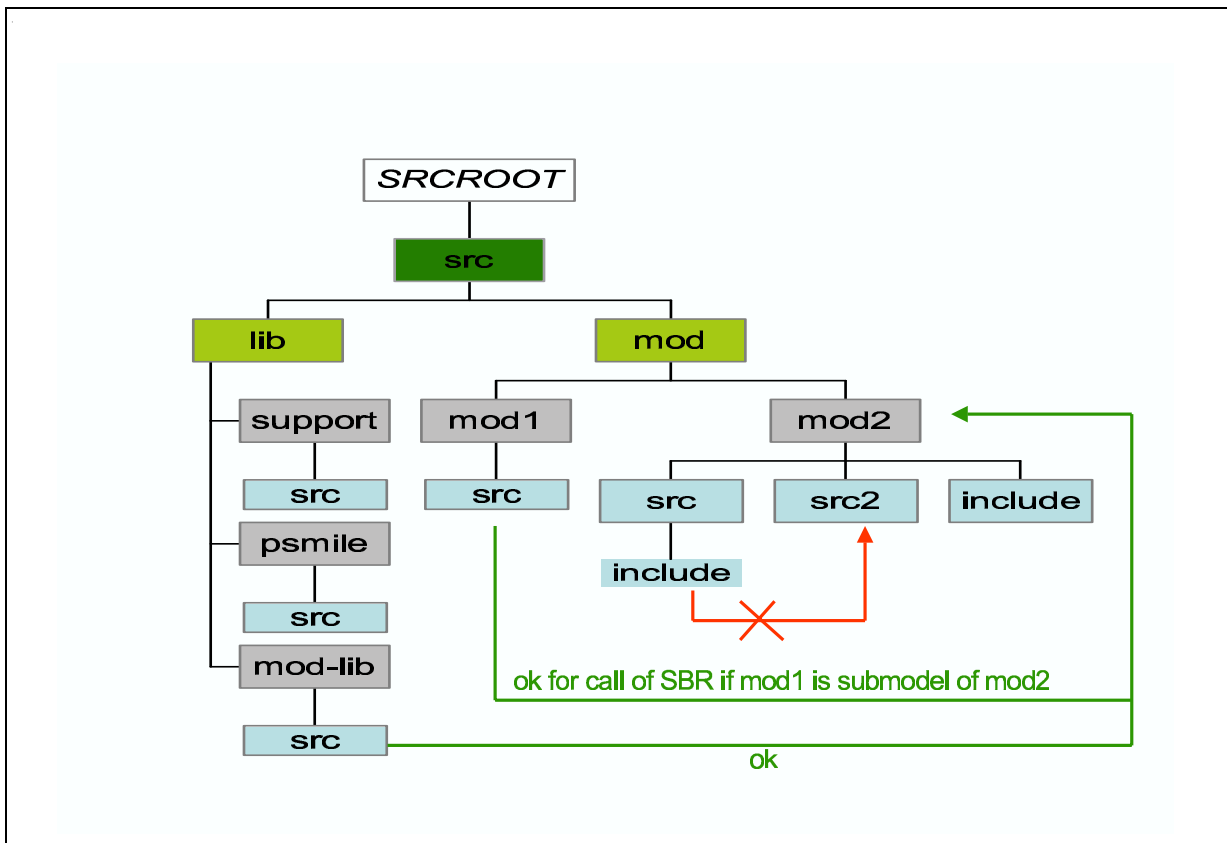


Figure 1.2: PRISM SCE source code directory structure. The USE of FORTRAN90 modules of libraries is allowed as well as those of submodels and CALLs of their subroutines (green arrows). If the call of a submodel is optional, CALL and USE statements should be wrapped by `cpp` flags. Header files are not allowed to be included from below another source code directory (red arrow).

A somehow different situation is the inclusion of calls of routines from libraries and submodels. For the submodels, it means that with the source code of the calling model that of the submodel has to be available. Since a model may potentially be coupled to a large number of submodels, and also due to license restrictions, it may not always be possible to provide all these source codes. A solution is the wrapping of optional source code from submodels or libraries with `cpp` flags. Doing so, the code is not known to the compiler nor to the loader.

It remains however, a problem with the (`g`)make based compile tools. The script `Append_Depend-`

encies which searches for the prerequisites of a compile target does not take into account conditional compiling. Therefore, a source code file which contains the call to a subroutine of a submodel will unconditionally be included in the list of prerequisites. When the model is compiled (`g`)make searches for this prerequisite and stops if it is not found. This occurs even if it will not be used in the compilation which follows. In order to avoid such a situation, a model may contain in addition to its source code directories a directory named `make_dummies` containing files with the names of the prerequisites (in most cases `*.o` files or header files therefore). This directory appears in the list of directories of the `VPATH` (`g`)make variable after the directory which contains the 'real' files. It will therefore be found by the (`g`)make search only if the first (e.g. the submodel directory) does not exist in the local source code directory tree. Note that the `make_dummies` directory files should contain code that can not be compiled. The compilation is then stopped if it is interpreted by the compiler by some mistake. This `make_dummies` directory may also contain platform dependent source code files which are not available for all platforms.

A simple source code tree for a main model and a submodel is shown in Figure 1.2. It also illustrates the requirement to keep the source code of a model separated from that of other models.

root/src/lib

Libraries which shall be used by any model, as e.g. the PRISM PSMILe library have their source code stored in the branch `root/src/lib` in a subdirectory named after the library, e.g. `psmile`. Just as the component model directory names, the library names are always in lower case letters. Some of them are libraries which are installed on most platforms and sometimes in optimized versions. In such a case the pre-installed libraries should be used. This is e.g. the case for the `blas`, `linpack`, and `lapack` libraries. Other libraries are libraries which entered the PRISM system with a coupled model, however are used by more than one component model. An example is `ioipsl`, the I/O library of the IPSL (Institute Pierre-Simon Laplace) models. These libraries are stored in the `lib` branch of the `root/src` directory since no model is allowed to use source code from another model (see Figure 1.2).

All libraries of the present release have just one source code directory named `src`. This directory contains the `Makefile`. Unlike for the component models there is no base `Makefile_1` and no support to generate the prerequisites for the compilation.

Each model has a list of libraries which are linked to the model when it is loaded into an executable. Before the model is compiled it is checked whether these libraries are up-to-date and the necessary parts of the libraries are recompiled. Therefore, when the PRISM models are installed on a platform, the libraries are compiled first and should not depend on any source code of the models.⁴ There is an exception allowed for the OASIS libraries which use FORTRAN90 modules from OASIS.

If modules from libraries are included during compilation they will be searched in the respective build directory of the library. Binaries (i.e. `.o` files of c- and FORTRAN routines) are taken from library archives in `root/arch/lib` of the build directory (see 1.1.2). Header files may reside in a directory `include` at the same level as the `src` directory or below it. Documentation may be stored in a directory named `doc` (see Figure 1.2).

1.1.2 The *root/arch* directory

The `root/arch` directory is created at compile time. `arch` is the architecture name specified in the Operating System (OS) depending part of the compile scripts. It contains the compiler output. This directory is not stored in the PRISM CVS repository. Figure 1.3 depicts its structure. The `root/arch` directory contains three subdirectories `build`, `lib`, and `bin`.

⁴This would also prohibit the automatic generation of `Makefiles` planned for future releases.

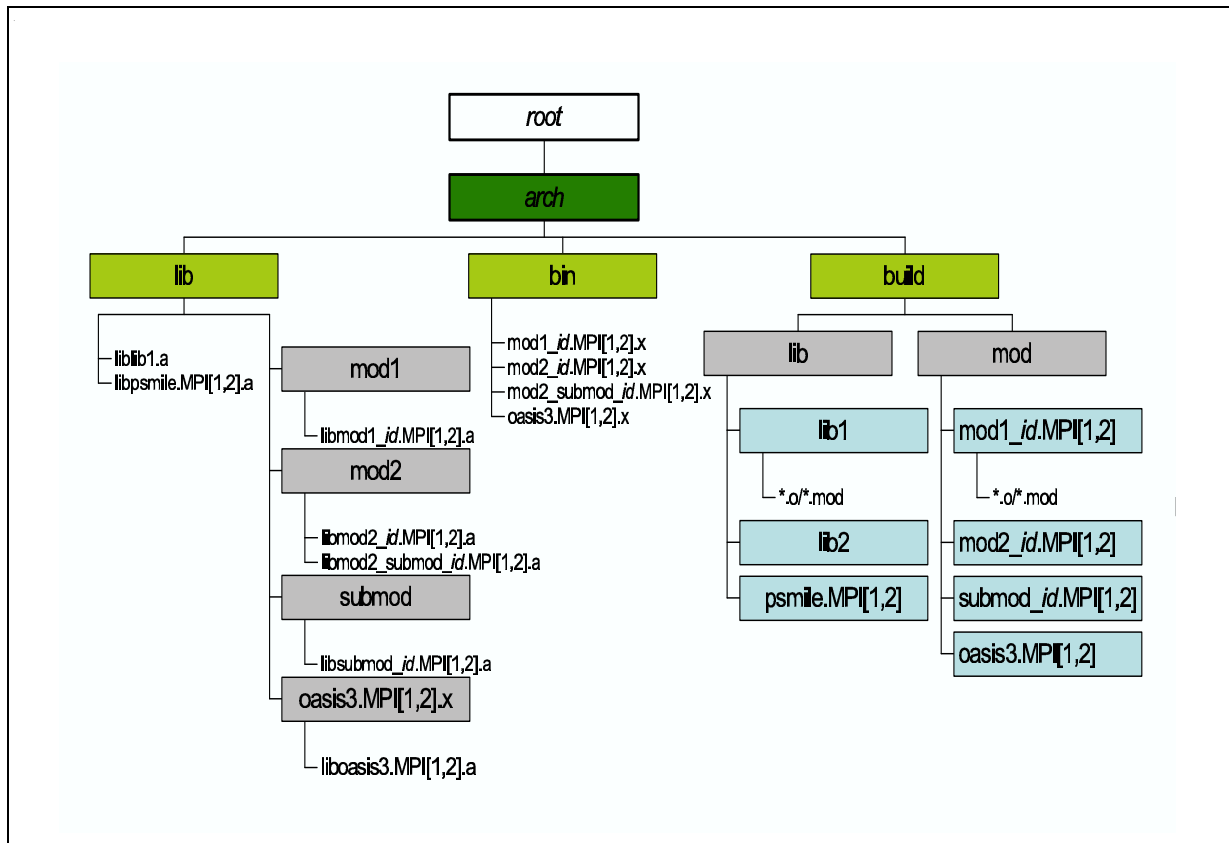


Figure 1.3: PRISM SCE directory structure containing the platform dependent compiler output

root/arch/build

From the *build* directory, similar to the *root/src* directory, two subdirectories branch out, the *mod* and the *lib* directories. Each of them have a number of subdirectories for the models and for the libraries. These are the directories where the models and libraries are built, i.e. the compilations are done. It contains therefore loadable binaries (*.o*) and the modules (*.mod*).

- *root/arch/build/lib*

The *lib* subdirectory contains one or two directories for each library. Two directories may exist for a library which contains conditionally compiled code depending on whether MPI-1 or MPI-2 is used, i.e. whether the models are spawned by OASIS3 or launched by the MPI unit. These are the communication libraries *clim* and *psmile* of the PRISM software. Their build directory has the suffix *MPI1* or *MPI2* appended to its name which is the library name in lower case letters. A separate directory for the compiler output is created for each option, since MPI-1 or MPI-2 may simultaneously be used at a single site. In the actual release there is no other library depending on the MPI-1/2 version. The only other configurations of libraries allowed are platform dependent features which are used unconditionally activated at a single site.

- *root/arch/build/mod*

The *mod* subdirectory contains one or more subdirectories for each component model after compilation. The component models have the *MPI1* or *MPI2* suffix unconditionally appended to their name even if they contain no MPI-1 or MPI-2 library dependent source code. This is done for maintenance reasons. It allows to upgrade models which are not yet able to run MPI-parallelized without being spawned by OASIS3 (this is the MPI-1 option) without need for changes of the SCE. In addition each model directory is given a tag which allows to discriminate between different versions at compile time or to tag the model for a specific experiments.

root/arch/lib

The `lib` directory contains the PRISM software library archives, other library archives, and archives of the component models. The latter are tagged with the version ID and the MPI library they will be used with, similar to the build directories. The libraries of the component models are all contained in a subdirectory named as the model. The other libraries reside directly in `lib`

root/arch/bin

After compilation all executables are stored in the directory `bin`. When the experiment execution scripts are prepared it must be specified as the place where the run scripts search for them. Executable names are built from the component models they contain, a tag to discriminate between different versions of the executable (e.g. resolutions, parameterizations chosen at compile time by `cpp` flag definition), and the `.MPI1/2` suffix. If a model and a submodel are assembled into the same executable they must have the same version tag associated at compile time. All executables are given the `.x` suffix in addition. An example is `root/arch/bin/model_submodel_ID.MPI2.x`.

1.1.3 The *root/util* directory

The `root/util` directory contains the code used for the generation of compile or run scripts and is accordingly separated into two branches, the `compile` and the `running` branches (Figure 1.1). Only the first is subject of this handbook, the contents of the `running` branch is described in Gayler and Legutke (2004). In addition the directory contains the script `COMP_lib.node` that is launched for the compilation of libraries either manually by the user or by a model compilation script. This script is not downloaded however from the CVS repository but has to be generated for the actual site (`node=host name`).

root/util/compile

The `root/util/compile` directory contains the perl script `Append_dependencies` which determines the prerequisites for the compile targets in `Makefiles` and a directory `frames` with subdirectories `include` and `include_node` with the code fragments from which scripts are assembled. In addition, this directory contains the scripts by which the compile scripts for models and libraries are generated, `Create_COMP_models.frm` and `Create_COMP_libs.frm`. The functionality and usage of these tools are described in Chapter 2.

1.2 Coding conventions: models and libraries

Coding conventions for FORTRAN and c(++) source code are described in Mangili et al. (2003).

When adapting a component model to the PRISM software it should be kept in mind that the coding should allow for a flexible exchange of the component model in different coupled models.

Therefore, if explicit units for FORTRAN I/O are used in a component model, these should not be hard-coded in the program. Instead they should be modifiable through namelist input. Otherwise unit conflicts may occur if more than one component model is assembled in one executable. It is contrary to the PRISM design of flexible coupling if such conflicts cannot be resolved without changing the model source code.

The ease of change of model resolutions, which is also an aim of PRISM, is increased when default parameters and switches for different set-ups (resolutions, grids, domains) are set automatically. If the executable depends on the model grid this can be done by conditionally compiled source code. If the

executable does not depend on the grid the parameters may be set according to the grid size once this is known.

Data which are used in many subroutines should be declared in a separate module in order to minimize the amount of recompilation for changes outside the declaration part of the code.

Configuring source code

The configuration of component models for different setups (grids, coupled constellations) with preprocessor flags (cpp flags) is supported. The naming convention for all cpp flag symbolic names is that they should start with two underscores (*__symbol*). The reason is that modules, subroutines, or variable names in user written source code rarely start with two underscores. When used in comments they should be cited as "cpp flag *symbol*" in order to prevent unintentional replacement of these names in the comments.

Note that (g)makee will not recognize a source file as 'modified' if only the state of cpp flags changes.⁵ In such a situation no automatic recompilation of the file will be done. Therefore, either the build directory of the model should be emptied (by calling the compile script with target `clean` (see Chapter 2) which implies a full recompilation, or the user detects manually which are the files concerned and deletes the corresponding *.o files. It is recommended to use conditional compilation only when it is unlikely that the user wants to switch from one definition state of a cpp flag to another frequently.

It follows a list of cpp flags already used in some models for special purposes related to coupling or other features of the PRISM system. When a component model is going to be newly adapted to the PRISM software, it is recommended to use these names for the same functionalities.

- `__coupled`: if different code must be compiled for the standalone model setup and a coupled model setup.
- `__prism`: if different code is present and must be compiled for use of the PRISM coupler and other coupler(s)
- `__oasis3/4`: if different code is present and must be compiled for use of the PRISM couplers OASIS3 or OASIS4.
- `__cpl_partner_model_name`: if special code must be compiled for coupling with the component model *partner_model_name*.

Note however, that cpp flags should not appear in libraries in order that the use of the same library version by all models is not prevented. An exception is the use of conditional source code for different platforms. The following cpp flags are used in the libraries:

`use_comm_MPI2`: defined when the MPI-2 library is available on the machine.

The PRISM default option is to use MPI-2 message passing for the communication between the models. If this library is not available, MPI-1 can be used (`use_comm_MPI1`; see Valcke et al. (2004a)).

`use_key_noIO`: defined when the `mpp_io` library will be not be used for output of model diagnostics with PSMILe calls.

`use_LAM_MPI`: defined for conditional use of the LAM MPI implementation

`NC_DOUBLE`: defined for double precision use of NetCDF in the `ioipsl` library

If platform dependent source has to be included in a component model or a library this should be wrapped with cpp flags as e.g.:

`__sgi`: for SGI platforms

⁵It is planned to include a method to detect cpp flag changes and appropriate reaction of (g)make at a later release of the SCE.

- `_SX`: for NEC SUPER-SX platforms
- `_vpp`: for Fujitsu VPP platforms
- `_Linux`: for Linux workstations.

For other platforms the symbolic name should be defined accordingly. Before a new name is invented the user should try to find out whether there is already a name defined which can be used. Platform dependent `cpp` flags can be found in the header files

`root/util/compile/frames/include_node/OSspecific_node.h`.

It is recommended to use these symbolic names. However, in many cases even in PRISM software the convention is not kept due to the fact that the code was written before the rules were specified.

Source code file names

Source code file name suffixes must be the standard suffixes which can be interpreted by compilers, i.e.

- `.F90` for FORTRAN free format source code which may contain preprocessing directives
- `.f90` for FORTRAN free format source code without preprocessing directives
- `.F` for FORTRAN fixed format source code which may contain preprocessing directives
- `.f` for FORTRAN fixed format source code without preprocessing directives
- `.c` for `c/c++` source code

Only these files will be used for the generation of the list of objects which are needed for compilation of the model.

Header files for inclusion into source code by a preprocessor should end with `.h`.

The tools for compilation require that a file containing a FORTRAN MODULE has the same name as the MODULE with suffix `.f90` or `.F90`. This implies that each file may contain only one MODULE. Otherwise the script which generates the prerequisites for the compiler rules may not work correctly.

1.3 Coding conventions: scripts

Some conventions for writing shell scripts have been defined. They are listed below.

- Use Korn shell scripts.
- Every script should have a comment block in the beginning to explain:
 - Name of the script; Useful when printed
 - Purpose: Some short explanation what the script is doing
 - Usage:
 - Parameters: Description of the parameters for the script.
 - Author: Who wrote or is responsible of the script

The most useful comment in a script is the usage printed by it when called with wrong arguments and/or with wrong options. Use box comments to comment a set of steps instead of commenting individual commands.

- Do not use side comments.
- The `-e` shell flag must be set.
- The `-u` flag to trap the use of undefined variables must be set.
Define empty variables by `variable=""`.
- All commands executed which are of general interest should be echoed into the output by `set -x`.
- Do not use any non portable scripting command.
- Use all lower case letters for directory names.

- Use all upper case letters for all exported variables.
- Use all lower case letters for all local variables.
No mixed case variables are allowed. Underscore is allowed except as the first letter.
- Variable names should be short, but not cryptic. PRISM meta data conventions should be used when possible (see Carter et al. (2004)).
- Use indentation (2 byte) for control constructs such as `if / while / for` only. Longer indentations result in too long command lines.
- For readability do not use long lines (more than 80 chars) but structure lines logically with continuation lines.
- Use here-document to a command rather than creating a separate file (if possible).
- Most commands accept input in so called free format which allows the input to be indented. The end of file indicator should not be indented. All options to a command should be separated (command `-a-o` rather than `command -ao`).
- Do not call a program by its absolute path; use `PATH` variable or relative paths.

1.4 The PRISM CVS repository

The latest version of the PRISM system or any released and tagged version can be downloaded from the central PRISM source code repository. The repository is based on the Concurrent Versioning System (CVS). It is accessible via a web browser⁶ or by direct access methods (`'pserver'` method with password authentication or `'ext'` method via `ssh` connection) from selected PRISM sites.

Most of the PRISM software is open source and can be downloaded without restrictions as user `'guest'`. To obtain the password please contact `prism_help@enes.org`.

Exceptions are the monitoring SMS system (Constanza et al. (2004)) and a few of the component models. License forms for Earth System component models or the SMS system can be downloaded from the table of CVS content maintained at the PRISM web site. They have to be sent to the corresponding model or system administrator indicated on the form. A user name and password for the download will then be made available.

On a server which allows direct access to the CVS repository, a CVS session is started with `cvs -d :pserver:guest@bedano.cscs.ch:/users/cvs login`.

Alternatively, when the `CVSROOT` environment variable is set (`setenv CVSROOT :pserver:guest@bedano.cscs.ch:/users/cvs` in a `c-shell`) a session is simply started by `cvs login`.

The source code and tools are downloaded by typing `cvs checkout [-r tag] module` for the release of the CVS module `module` tagged with `tag`. If no tag is specified, the latest versions will be downloaded.

The CVS session is ended with `cvs logout`.

CVS modules

The download of PRISM models is organized by CVS modules.

The CVS module to download a component model is the model name in upper case letters. Such modules comprise everything needed to compile the corresponding component model, i.e. the model source code, all library source codes linked to the model, and the tools to compile it, respectively those for the creation of scripts for compilation and execution.

⁶<http://prism-cvs.cscs.ch/cgi-bin/cvsweb.cgi>

For coupled models (this may also be a standalone model in this context, i.e. a model consisting of one component model only) there are three modules,

CPLMOD: sources of all component models, associated libraries, utilities and the input data tar-file

CPLMODSRC: sources of all component models, associated libraries and utilities

CPLMODDATA: the input data tar-file

For a coupled model consisting of more than one component, *CPLMOD* is a name depending on the component models participating in the coupled model. A table with coupled model names is found at the PRISM web site.

If e.g. the CVS module TOYCLIM is downloaded, the source code of the OASIS3 coupling software, of the three toy models TOYATM, TOYOCE, and TOYCHE, and the tools for the generation of compile and run scripts, as well as input and some example output data are obtained.

The software is installed in the PRISM root directory ``pwd`/prism`, i.e. in a directory `prism` (created) in the directory where the download has been triggered. Detailed instructions⁷ on how to download a PRISM model and the content of the repository can be found on the PRISM web site.

⁷<http://prism.enes.org/Portal/Downloads>

Chapter 2

Compiling

All compilation is based on the GNU (`g`)`make` tool. The scripts and input files are set up in a way that a minimum of recompilation is done when something in the source code has been changed. Therefore, for each binary which has to be compiled, a list has to be given of all prerequisites that have to be available and up to date for a correct compilation. This is specified in the input files to the (`g`)`make` command. The default name for this file is `Makefile` which is also the name used in the SCE. One makefile has to reside in each of the `/root/src/mod/model` and `/root/src/lib/library` source code subdirectories containing compiler input files. Some directories, e.g. those containing only input files for the preprocessor (`include` directories) do not need to have a makefile.

The makefiles in the model or library source code directories are called low level makefiles. They are described in Section 2.1.1 for the component models and in Section 2.2.1 for the libraries. Low level model makefiles are generated semi-automatically. How this is done is subject of Section 2.1.2. There is no support yet to generate the library low level makefiles.¹

The low level makefiles are fully portable. All non-portable specifications are set in a shell script, called compile script, and are exported to the makefiles. There is exactly one compile script for each component model. Model compile scripts are always specifically created for a particular component model as well as for a specific computing site and platform. How this is done is described in Section 2.1.3. The structure of these scripts is also described there.

There is only one compile script for all libraries. The structure of this script and its generation is described in Section 2.2.2. It is explicitly generated for each site.

The low level makefiles of both models and libraries, trigger the (re)compilation of all files with standard FORTRAN or c(++) source code file names in their directories. All compiler loadable output binary files are archived into a single archive for each component model, called the model library, and for each library. The models may have different archives for differently configured versions, while the libraries are not allowed to have different versions on one platform². When an executable is loaded, the libraries needed must be provided as input to the loader. The compile scripts of the component model as well as that of the coupler therefore check whether all of their libraries are up to date. This is done by launching the script for library compilation. If there are other library prerequisites for a component model, e.g. submodel libraries (see below), these are checked as well before the compilation of the model itself is done.

¹This is planned for a future release.

²Exceptions are the PRISM communication libraries `PSMILe` and `clim` which are configured for the method of model launching (see Section 2.2.2).

2.1 Model compilation

The PRISM project has defined 6 classes of component models. These are atmosphere general circulation models (AGCMs), atmosphere chemistry models (ACM), ocean general circulation models (OGCMs), marine biogeochemistry models (BGCMs), sea ice models (SIMs), and land surface schemes (LSSs). A set of source code files representing one of these subsystems of the Earth System is therefore considered to be a component model and has to be stored in its own model source code directory in `/root/src/mod` in the PRISM SCE as described in Chapter 1³. Each of the component models is compiled separately.

A component model is called a main model if it is loaded into an executable using an entry point in one of its own binaries. When a model library is linked to a main model, the model is called a submodel of that main model. A component model can have the potential to be both a main model or a submodel. Whether it is a main or a submodel is decided when the executable it belongs to is loaded. Besides this last loading step all component models are treated similarly. The OASIS3 coupler is treated as a component model as well.

Each component model being part of a coupled model needs to have its own compile script. However, if a coupled constellation is made up of more component models than there are model executables, that is if some models are used as submodels, only the compile scripts of the main models which create their own executable need to be launched by the user.

The main model scripts will launch the compile scripts of their submodels. In addition, all component model compile scripts launch the library compile script with the list of libraries they need. This chain of compile script launching is depicted in Figure 2.1.

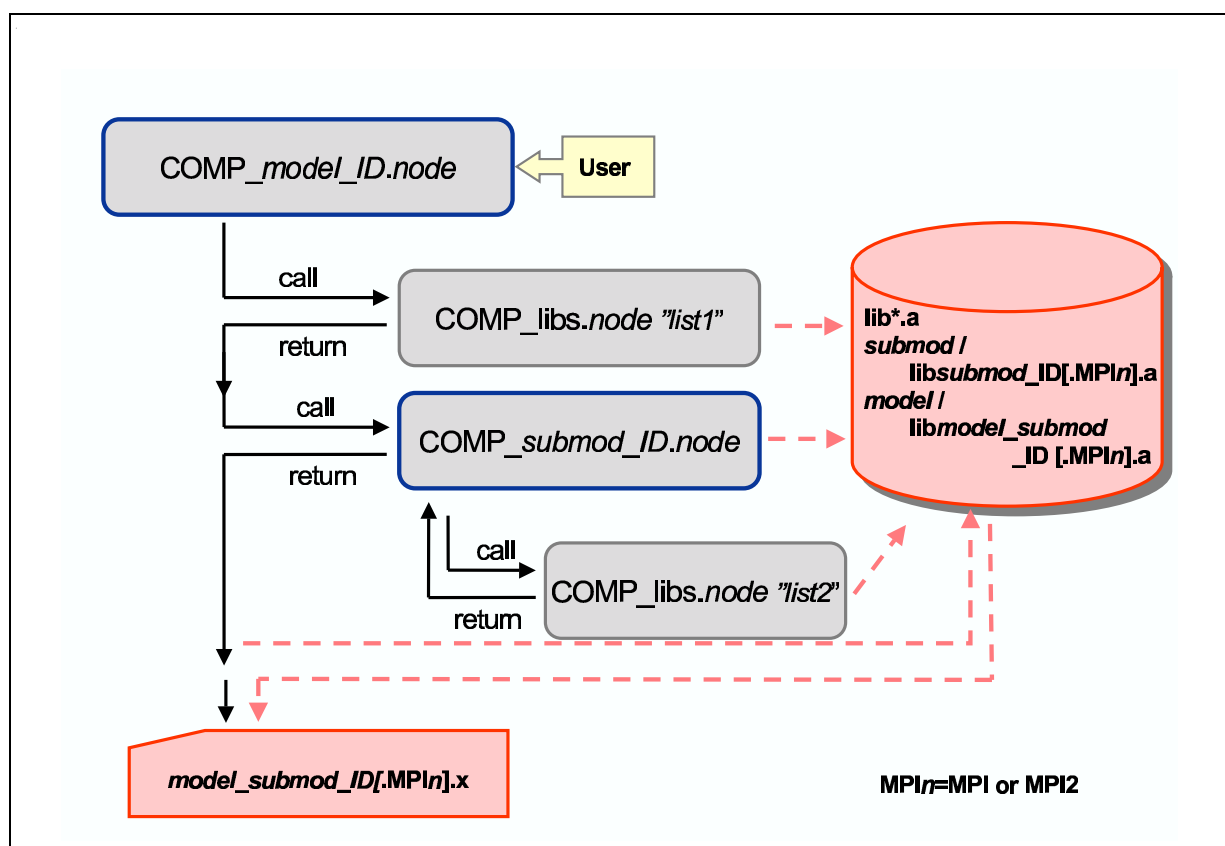


Figure 2.1: Chain of scripts that is run for the compilation of a main model calling a submodel. The gray boxes represent the scripts, the red symbols represent the final output of the process, i.e. the executable, and the binary archive libraries. Script flow is indicated by full arrows. The dashed arrows show I/O to/from disk.

³This is recommended also for a package of model physics parameterizations which is intended to be used by more than one component model, or if it is planned to exchange the package by another one.

Note that the submodel compile script and the main model compile script have the same version ID in Figure 2.1. This is required by safety reasons. The component models may need to be configured for the constellation by the preprocessor. Component models configured for the same constellation are assumed to have the same ID. Figure 2.2 displays the error message printed by the AGCM LMDZ compile script when no compile script for its submodel, the LSS ORCHIDEE, with the appropriate ID is available.

```

:
Checking the submodel orchidee
--- ../src/mod/orchidee/COMP_orchidee_I01.ds must be created first! ---
--- Use ../util/compile/frames/Create_COMP_models.frm. ---
--- This script is stopped!

```

Figure 2.2: Error message output by the LMDZ compile script if no script for its submodel ORCHIDEE with the required version ID (I01) is available. The script prints the full site dependent path. It is replaced by ‘...’ in the figure.

Consistent configuration and tagging of all components of a coupled PRISM model is automatically done if the compile scripts are generated with the tool `Create_COMP_cpl_models.ksh` (see Section 2.1.3). Note that all compilation, whether of libraries or of main or sub models, first creates a compiler output binary archive. These archives are used by the loader to build the executable. The executable that is generated has both the main model and the sub model names in its file name, together with the version ID and the message passing option (Figure 2.1). This is the default name used in the experiment run scripts of the SRE (Standard Run Environment).

Figure 2.3 displays the high level design of the compilation process with some more details for the simpler case of a component model without calling a submodel.

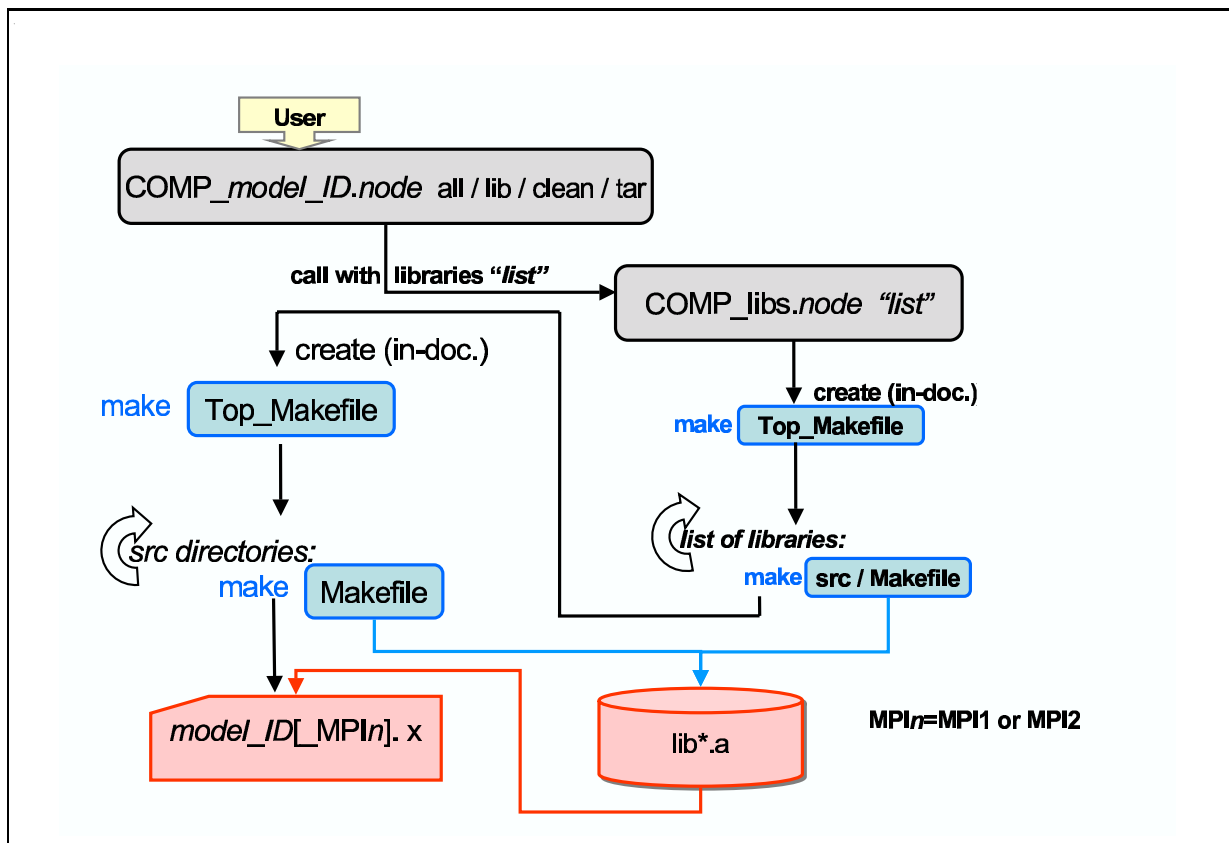


Figure 2.3: High level design of a component model compile script. Gray boxes are scripts, blues ones represent the makefiles, the red ones the final result.

A component model’s compile script can be called with four targets: `all`, `lib`, `clean`, and `tar`. If the model hosts a submodel, the compile script of the latter will be called (with the same target) only if the target is `all` or `clean`.

With the default `all`, the full job is done, that is, the model’s source code is updated, its libraries are

updated, and if necessary an executable is built.

If `lib` is specified, only the model's binary archive library is built, no attempt to create an executable is made. A submodel compile script is not called. If the model has no main program, the `all` target is redefined to become the `lib` target, i.e. whether the script is run with `all` or `lib` makes no difference.

With `clean`, the model's build directory is cleaned from all compiler output and makefiles which are stored there during compilation. In addition the model binary archive library is deleted. If the model is configured for a constellation where it contains a submodel, the submodel's binary archive library is also deleted since its compile script is launched with `clean` as well. No cleaning of other libraries is done. A rebuilt of general libraries can be enforced by calling the library compile script directly with the `clean` parameter (see Section 2.2.2).

Finally, the script can be called with the target `tar`. This creates a tar-file with all source code of the model, of its libraries, and all utilities of the SCE and SRE.

Each script first launches the compile script to update its libraries. Note that the library compile script has no ID in its name. The same libraries are used for all component models. They are not configurable. The library updates have to be done first, since the models can USE FORTRAN90 modules of libraries and therefore the modules of libraries may be prerequisites to the model and have to be done first. The USE of model FORTRAN90 modules in libraries is not allowed.

A top level makefile is created then, called `Top_Makefile`. It defines the rules for the targets which are passed to the compile script when it is called (see Fig. 2.12 and 2.3). The rule for targets `all`, `lib`, and `clean` is to loop over all source code directories and call the low level makefiles therein with that target. The default action of the low level makefiles is to update the model library archive. In one of the source code directories, where the main program is stored, the model executable is loaded. This directory is updated the last.

2.1.1 Low level model makefiles

The model low level makefiles reside in the source code directories of the models. Each directory with input to the compiler needs to contain exactly one makefile. The low level makefiles are fully portable and are therefore downloaded from the CVS PRISM repository with the models. They consist of two parts: a first part contains the general specifications such as search paths or rules for compilation, and a second part contains the prerequisite lists of the binaries to be compiled.

The first part of an example low level makefile is listed in Figure 2.4.

With this makefile `(g)make` first generates lists of `.o`-targets of the actual source code directory. Each file in the directory having one of the suffixes allowed for compiler input (`.F90`, `.f90`, `.F`, `.f`, and `.c`, see Section 1.2 of Chapter 1) is taken to be a prerequisite for a target to be built. A pair of lines is present in the makefile for each of these suffixes. The first line searches for the input files, the second replaces the suffixes by `.o`.

The variables `OBJS1` to `OBJS5` contain then a list of binary prerequisites for the model library target (`$LIBRARY`). The name of the library is set in the compile script of the component model and exported to the makefile.

The submodel name is included in that name if a submodel is called by the model in the sense defined by the SCE (see Section 2.1). Also, the `MPI[1,2]` string appears in the library name if the model runs in a constellation including OASIS3. Both are part of the model library name because the model may be configured by the preprocessor for the coupled constellation and for the MPI usage. Since a change of `cpp` flags is not recognized as a change of model code, `(g)make` would not trigger a recompile if the use of a submodel or the MPI version is changed. All other compiler configuration is subsumed under the version `ID`. It is in the responsibility of the user to enforce the necessary recompilation if any other configuration by `cpp` flags is changed.

It follows in the low level makefile a line with the specification of the `VPATH` variable, which has to contain a list of directories where any of the commands of the makefile search for prerequisites. In the example of Figure 2.4 these are (see also Figure 1.1 of Chapter 1):

1. the source code directory `$(SRC)` containing the makefile
The name of this directory is exported from the compile script. It contains the prerequisites for the `*.o` compiler output files. Files with any of the suffixes `*.c`, `*.f`, `*.F`, `*.f90`, or `*.F90` will be taken as input files. Therefore, there may not be two files with the same base-name but different suffixes. All such files have to contain code suitable for compiler input.
2. a subdirectory `include` of the source code directory with input for the preprocessor
This subdirectory may only contain header files for inclusion by a preprocessor into files of its parent directory
3. a directory called `include` with input for the preprocessor at the same level in the model directory.
All header files potentially included into files of more than one of the source code directories should reside there.
4. the built directory for the PSMILe library
This directory is a search directory since `MODULEs` of the PSMILe library are `USED` by subroutines of the component model. The corresponding directory in the OASIS3 makefile is `clim` which is used by OASIS3 instead of PSMILe for the communication with the models.
5. 3 directories with include files of libraries installed by the system (not PRISM) administrator for the preprocessor
These directories contain include files for interfacing with the NetCDF and the MPI libraries, and other system include files.
6. the directory containing the `model` libraries
It is a search directory since the model library is either the final target or it is a prerequisite to the loader command.
7. a directory containing the `submodel` libraries
The component model may host a submodel which has to be linked by the loader and therefore `(g)make` will search for its library. The present release allows for only one submodel⁴. The submodel library's name is contained in the `DEPLIB` variable which is set in the compile script and exported to the makefile.
8. the directory containing other PRISM libraries
9. a directory containing dummies for `(g)make` prerequisites
These dummies are needed if the situation occurs that a prerequisite is not available to `(g)make`, however, it appears in a prerequisite line for a `(g)make` target. This may happen e.g. if `cpp` include instructions in the source code are wrapped by a `cpp` flag which make the preprocessor to skip the file and the compiler to ignore it. The tool which checks for dependencies, however, will detect it and include it into the second part of the makefiles. Since it is not used for compilation but only by `(g)make` it should not contain correct compiler input (see Section 1.1 of Chapter 1)⁵.

Note that this first part of the model low level makefiles does not contain any variable specific to the model. It looks therefore about the same for each source code directory of each model.⁶

All paths in the makefiles are given relative to the build directory where the compilation is done (`/root/arch/build/mod/model_ID.MPI[1/2]`, see Section 1.1.2 of Chapter 1).

⁴This can easily be extended to allow for more than one submodel.

⁵Obsolete feature.

⁶Specific features are entered mainly when the model does not fully comply to the SCE conventions and need some extra specifications. This will prevent a fully automatic generation of the makefile which is planned for the future. If such extra code is needed it should be checked therefore whether the model can be made PRISM SCE compliant instead with reasonable effort to avoid the extra code in the makefile.

The next lines of the makefile (Figure 2.4 define name and path of the model library, of the executable, and of the include directories, one for the FORTRAN compiler and one for the c++ compiler. The FORTRAN compiler include variable has two ways of setting the option name: `-I` for inclusion of header files by the preprocessor or compiler and `$(I4mods)` for the specification of directories with FORTRAN MODULES to be USEed by the FORTRAN90 compiler. The value of `$(I4mods)` is passed from the OS specific part of the compile script. The program (executable) name is also set in the model compile scripts. It depends on the model name, the submodel name, the specification of the MPI library used, and a version tag subsuming all other configuration.

The next lines define the targets and the rules to make them. There are three main targets which can be passed to the `(g)make` command by the user:

`clean`: Called with this target, all compiler-generated files in the model's build directory as well as the model library archive is deleted as required by the rules following the target definition.

The `Top_Makefile_*` that have accumulated since the last cleaning action are also deleted.

`lib`: With this target only the model library `$(LIBRARY)` is created. There is no attempt to create an executable file by the loader.

`all`: This is the default target to every `(g)make` command. It makes the executable. If the model is not a potential main model, i.e. if it is not designed to use an entry point of its own, this target is defined as `all:$(LIBRARY)`, i.e. it is equivalent to the `lib` target.

```

SRCS1 = $(shell find ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC)) -name '*.F90' -print)
OBJ1 = $(patsubst ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC))/%.F90, %.o, $(SRCS1))
SRCS2 = $(shell find ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC)) -name '*.f90' -print)
OBJ2 = $(patsubst ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC))/%.f90, %.o, $(SRCS2))
SRCS3 = $(shell find ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC)) -name '*.F' -print)
OBJ3 = $(patsubst ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC))/%.F, %.o, $(SRCS3))
SRCS4 = $(shell find ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC)) -name '*.f' -print)
OBJ4 = $(patsubst ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC))/%.f, %.o, $(SRCS4))
SRCS5 = $(shell find ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC)) -name '*.c' -print)
OBJ5 = $(patsubst ../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC))/%.c, %.o, $(SRCS5))
VPATH = .:\
../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC)):\
../../../../src/mod/$(MODEL_DIR)/$(strip $(SRC))/include:\
../../../../src/mod/$(MODEL_DIR)/include:\
../../lib/psmile.$(CHAN):\
$(NETCDF_INCLUDE):$(MPI_INCLUDE):$(SYS_INCLUDE):\
../../lib/$(MODEL_DIR):\
../../lib/$(SUBMOD1):\
../../lib:\
../../src/mod/$(MODEL_DIR)/make_dummies
LIBRARY = ../../lib/$(MODEL_DIR)/lib$(MODLIB).a
PROG = ../../bin/$(EXEC).x
INCL = -I../../src/mod/$(MODEL_DIR)/include \
-$(I4mods)../../lib/$(MODEL_DIR)/include
INCLSC = -I../../src/mod/$(MODEL_DIR)/include
clean:
rm -f $(LIBRARY) i.* *.o *.mod Top_Makefile_*
all: $(PROG)
lib: $(LIBRARY)
$(PROG): $(DEPLIBS) $(LIBRARY)
$(F90) $(LDPLAGS) -o $@ $(MAINPRG).o $(LIBS)
$(LIBRARY): $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) $(OBJ5)
$(AR) $(ARFLAGS) $(LIBRARY) *.o
.SUFFIXES:
.SUFFIXES: .o .c .f .F .f90 .F90
%.o: %.F90
$(F90) $(F90FLAGS) $(INCL) -c $<
%.o: %.f90
$(f90) $(f90FLAGS) $(INCL) -c $<
%.o: %.F
$(F) $(FFLAGS) $(INCL) -c $<
%.o: %.f
$(f) $(fFLAGS) $(INCL) -c $<
%.o: %.c
$(C) $(CCFLAGS) $(INCLSC) -c $<

```

Figure 2.4: First part `Makefile_1` of a low level component model makefile. This part is largely independent of the specific component. The full `Makefile` consists of `Makefile_1` with the prerequisites for the compiler targets appended.

For potential main models it then follows the specifications to make the executable `$(PROG)`. Prerequisites are the model library as well as all other libraries linked to the model. The list of libraries to be checked (`$(DEPLIBS)`) is set in the compile script. It includes the `mpp_io` and `PSMILE` libraries to enable communication between executables and PRISM compliant NetCDF output, any submodel library if necessary, and other libraries. The program is made by linking the necessary libraries to the main routine `$(MAINPRG)` (next line). Loader commands and flags are OS specific and passed from the compile

script. The name of the program with the main entry point is given separately from the other compiler output binary files in the archive. This allows to have different targets with different entry points if needed. It is also passed from the compile script and is specified there in the model specific section (see Section 2.1.3).

The rule to make the model library $\$(LIBRARY)$ is just to archive all compiled binaries of the models build directory. Archive command and flags are again passed from the compile script because they are OS dependent.

The next two lines define the suffixes for $(g)make$. First, all implicit rules are invalidated. Then the suffix rule for making binary $*.o$ files from source code files with suffixes $.c .f .F .f90 .F90$ are declared.

Finally follow the rules to make the $.o$ -files for each 'class' of source code files classified by their suffix. A different set of compiler options can be passed from the compile script for each class. This provides the possibility to use different compiler options for different file classes. If additional distinction is needed, classes can be defined by using additional directories to store the source code.

The first part of the model makefiles just described has to be provided by the model developer and must be called `Makefile_1`. Most model low level makefiles show some deviations from the example makefile to allow for some special features of the model. If the model has more than one source code directories it may be necessary to include some of them into the makefile's `VPATH` or `INCLC(C)` variables. These directories are detected with the help of a `perl` script which is used to generate the lists of prerequisites for the binary files. How this is done is described below in Section 2.1.2.

2.1.2 Generating low level model makefiles

The `/root/util/compile` directory contains a script `Append_dependencies` which can be used to generate the prerequisites for the compiler output binaries for any source code directory of component models adapted to the SCE.⁷

The script may be run from anywhere in the PRISM directory tree. It has a help-function which can be activated with either `'-'` or `'-help'` or by giving the wrong number of parameters (Figure 2.5).

```
Append\_dependencies --help
*****
*
* This script must be called with 2 parameters:
* 1. model name (directory name of the model source code)
* 2. source code subdirectory for which to create the Makefile
* One or both of the parameters is empty!
* The script is stopped!
*
*****
```

Figure 2.5: Help text of the script `Append_dependencies`. The text is output either when help is required (with input `'--help'`) or when the script is called with less than 2 parameters.

The script requires two positional parameters: a model name and a source code directory name.

```
Append_dependencies ll 1
*****
* This script must be called with 2 parameters:
* 1. model name (directory name of the model source code
* (all lower case))
* 2. source code subdirectory (for which to create make
* dependencies)
* The first parameter is not a valid model name:
* model name = ll
* Valid models are : toyatm toyche toyoce echam5
* mpi-om oasis3 hamocc opa lim toy4opa arpege_climat4
* pisces mozart lmdz orchidee toy4arpege
*****
```

Figure 2.6: Message printed by `Append_dependencies` if an invalid model name is entered. It is a list of all component models installed in the local PRISM SCE directory tree.

⁷The script can not be used to generate the prerequisites for the library compile rules. This extension is planned for a future release of the SCE.

If two parameters are given, a `perl` script, created as a here-document by the `k-shell` script, is run. The `perl` script checks whether the first parameter is the name of a model directory installed at the actual site. If this is not the case it outputs the names of all available component models (Figure 2.6).

The script also gives support to specify a valid source code directory (Figure 2.7). When an invalid directory name is entered with a correct model name it outputs a list of subdirectories.

```
Append_dependencies lmdz src
*****
* Source root directory : /import/ds9b/ipf/k/k204020/prism_test/src/mod
* Model name           : lmdz
*****
* This script must be called with 2 parameters:
* 2. source code subdirectory
*   (for which to create Makefile dependencies)
*   The specified vakuue is not a valid directory name.
*   Valid directories are : grid bibio include dyn3d phylmd filtrez
*****
```

Figure 2.7: Messages printed by `Append_dependencies` if the directory entered with the model name is not a subdirectory of that model: list of all component model subdirectories (excluding 'doc').

If both model and source code directory names are valid, the script tries to read a file `Makefile_1`. The message printed if this file is not available is shown in Figure 2.8.

This is not necessarily a fault. It could also mean that the script should not be called for that directory. In the example of Figure 2.8 the script is called for a directory which only contains include files for the preprocessor and therefore does not need a makefile.

```
Append_dependencies lmdz include
*****
* Source root directory : /import/ds9b/ipf/k/k204020/prism_test/src/mod
* Model name           : lmdz
* Create dependencies for Makefile in source code directory : include
* No Makefile_1 in directory include !!
```

Figure 2.8: Messages printed by `Append_dependencies` if `Makefile_1` is not available

The script then appends to `Makefile`, for each input file to the compiler (files with suffix `.c .f .F .f90 .F90`), a line with the prerequisites. This is done by scanning these files for include instructions such as `USE` or `#include` or the like.

Models which have more than one source code directory may have inter-directory dependencies. That is, a prerequisite may reside in a directory different from the one being processed. The script would not prepare a list of prerequisites for it by just scanning the actual directory files. Therefore, for each prerequisite `*.o` not having a prerequisite list for itself yet, the script searches for a corresponding files with with suffix `.c, .f, .F, .f90, or .F90` in the other source code directories and prepares a list of prerequisite for it as well. How many iterations have to be done until there is no more prerequisite without prerequisite-line depends on the number of source code directories and the interdependencies of the files therein. The complete prerequisite lists are essential to guarantee a correct compilation in one go. No repetition larger than 3 was needed with the models presently in PRISM. The number can be modified in the script (variable `$times`). At the end, the script outputs the directory names which have to be included in the `VPATH` or the `INCL(C)` variables of `Makefile[_1]`. Note that, if any manual change is done to `Makefile` only, it will be overwritten by the next call of `Append_dependencies` if it is not done to `Makefile_1` as well.

2.1.3 Component model compile scripts

Each component model being part of a coupled model needs to have its own compile script.

The PRISM source code packages or CVS downloads come without compile scripts since these are OS and site dependent. Instead, tools are provided, which enable the user to easily create the scripts for PRISM sites⁸.

⁸A PRISM site in this context is a site for which OS and site specifications for the component model in consideration are provided with the PRISM software download.

Create_COMP_cpl_models.ksh

For a PRISM coupled model, all compile scripts of its components are conveniently created by a single call of the script `Create_COMP_cpl_models.ksh` (Figure 2.9). The script resides in the directory `/root/util/compile/frames` where the SCE utilities are stored.

The compile scripts it creates are configured for the platform and for the coupled constellation as specified by the input parameters. The first parameter, the coupled model name is the CVS module name (in lower case letters) by which the coupled model is downloaded from the CVS PRISM repository (Section 1.4). The second parameter is a version or experiment tag for the model executables, and the third parameter gives the node name of the PRISM compile platform if different from the one where the script is called.

```

Create_COMP_cpl_models.ksh ipsl_cm4 ID [node]
  launches →
/Create_COMP_models.frm lmdz "" - "" node ID "lmdz orchidee opa lim"
/Create_COMP_models.frm orchidee "" - "" node ID "lmdz orchidee opa lim"
/Create_COMP_models.frm opa "" - "" node ID "lmdz orchidee opa lim"
/Create_COMP_models.frm lim "" - "" node ID "lmdz orchidee opa lim"
/Create_COMP_models.frm oasis3 "" - "" node
/Create_COMP_libs.frm "" - "" node
  creates →
/root/src/mod/oasis3/COMP_oasis3_MPI2.node
/root/src/mod/lmdz/COMP_lmdz_ID.node
/root/src/mod/orchidee/COMP_orchidee_ID.node
/root/src/mod/opa/COMP_opa_ID.node
/root/src/mod/lim/COMP_lim_ID.node
/root/util/COMP_libs.node

```

Figure 2.9: Call of `Create_COMP_cpl_models.ksh`, the calls of the compile script generator `Create_COMP_models.frm` it triggers, and the list of generated compile scripts.

The script creates, by the appropriate calls to a second tool, `Create_COMP_models.frm` in the same directory, one compile script for each component model that is part of the coupled constellation and moves them into the respective PRISM SCE model directories (Figure 2.9). In addition to the model compile scripts, a script for the libraries linked to the models is created. Figure 2.9 displays these calls and the resulting scripts for the example of the IPSL coupled model `IPSL_CM4`, which is composed by the components `LMDZ`, `ORCHIDEE`, `OPA`, and `LIM`.

Create_COMP_models.frm

The script `Create_COMP_models.frm` may also be called for each component separately. It resides in directory `/root/util/compile/frames` and can be launched from anywhere in the local PRISM root directory.

Figure 2.10 displays its call and control messages for the example of the component model `LMDZ`.

When the compile script is generated without error, the script's control output starts with the model name, and the model version ID. This is followed by a list of partner models which will make up the coupled constellation and has to be passed to the script in the last command line parameter. The specification is needed in order to correctly configure the component model for the constellation by the preprocessor. If the script is generated on the machine where it will be run, the Operating System (OS) name of the machine and its node name are printed next. A compile script for a PRISM platform can be generated on any machine if the node name of the target machine is given. In that case a warning is output and the OS is not printed. Next, control messages relating to the active command line parameters (default or specified) are listed. These are the version of the message passing library (MPI1 or MPI2), and where the standard

```

Create_COMP_models.frm lmdz " " - " " " I01 "lmdz orchidee opa lim"
*
* Compile script for model lmdz, version=I01 is created
*
* Partner component models are: lmdz orchidee opa lim
* Creating compile script
*   on/for a Linux platform with node name ds8.
* Abbreviated node name is ds.
* The message passing will be MPI2.
* (g)make standard output will directed to screen.
* (g)make standard error will be directed to a file.
* The model directory is ../src/mod/lmdz
* The compile script name is ../src/mod/lmdz/COMP_lmdz_I01.ds

```

Figure 2.10: Control messages from compile script creation for component model LMDZ in the constellation IPSL_CM4 (with LSS ORCHIDEE, OGCM OPA, and SIM LIM). These messages are output if the script is run on the compile server and the compile script generation comes to a normal end. The script always prints the full site dependent path. It is replaced by '...' in the figure.

and error output messages will be directed to. The compile script, after generation, is automatically moved into directory `/root/src/mod/model`, where `model` is the name of a PRISM component model, a main model or a submodel. Component model compile scripts are called `COMP_model_ID.node` (last line). The suffix may be an abbreviated version of the actual node name.

The scripts can be called with up to seven positional parameters. A help function is prompted as shown in Figure 2.11. It prints options for these parameters and also adds the defaults at the end of the lines in brackets. If 'none' is displayed there, the parameter is required.

The first parameter is required and must be a component model name in lower case letters identical to the name of the directory where the model is stored. The 2nd parameter has MPI2 as default and may only be changed into a non-default value MPI1 or NONE. NONE has to be specified for 'standalone' models not needed to be linked to PSMILE since they do not run in a constellation with OASIS3. Parameters 3 and 4 define the standard and error output devices. The default is a file in the model directory. A minus sign stands for the screen. If the 4th parameter is a plus sign the error output is directed to the standard output device. If the scripts are to be created for a different machine the node name of that machine has to be given as it is used in the PRISM system to identify the OS and site dependent specifications in the scripts. This may be different from what is obtained by 'uname -n' if the latter is not unique (e.g. the node where the script will be run is not unique). Parameter 6 is also required and gives a version or experiment acronym used to tag the executables for use by the SRE to identify the right file. With the last parameter the list of component models which make up the coupled constellation has to be passed to the script. Configuration by `cpp` flags if needed will be done accordingly. An exception is the OASIS3 executable which is not configurable besides the MPI1/2 specification. Both the version acronym and the list of coupled partner models will be ignored for OASIS3 if given.

```

Create_COMP_models.frm --help
-----
This script may fail if you do not use the right version of m4!
The version you use is : GNU m4 1.4.1
Make sure this is GNU m4 younger than version 1.4!
-----
Usage
-----
$1=model name (none)
$2=""/"MPI1"/"NONE": message passing (MPI2)
$3=""/"-" : std output of compile script -> file / screen (file)
$4=""/"-" / "+" : err output -> file / the screen / stdout (file)
$5=""/"node name": node name of compile server ('uname -n')
$6="version acronym" (none)
$7="list of participating models" (none)
Specification of defaults is not allowed: write e.g.
Create_COMP_models.frm oasis3 " " - " " "
instead of Create_COMP_models.frm oasis3 "MPI2" - " " "
-----

```

Figure 2.11: Help text for the component model compile script creation tool. (none) indicates that this parameter has no default, a value is required. The others may be the empty string.

COMP_model_ID.node

Running a component model compile script *COMP_model_ID.node* (Figure 2.9) creates a binary archive for the component model in */root/arch/lib/model* and an executable in */root/arch/bin* if required (see Figure 1.3). Both directories are created during compilation if not present. Both the binary archive and the executable are tagged with the version *ID* and the message passing option (Figure 2.1). The name of the component model becomes also part of their names, and, if the model is configured to call a submodel, the submodel name is included as well.

This is the default name used in the experiment run scripts of the SRE (Standard Run Environment).

The compile script is created with all options set to default values (e.g. model physic parameterizations) and for a coarse resolution (for the case the executable depends on the model grid). All options can be changed by editing the script.

```
COMP_lmdz_I01.ds --help
This script runs on node ds8.
Model directory is ../src/mod/lmdz
----- Usage -----
$1=make_target      (all/lib/tar/clean; optional)
$2=message_passing (MPI1/MPI2; optional)
Default active values are:
make_target        : all
message_passing    : MPI2
-----
```

Figure 2.12: Help text of compile scripts. The script prints the full site dependent path. It is replaced by '...' in the figure.

A component model compile scripts provide a 'usage' help-function: when run with parameter '--help', the command line parameters are listed together with their actual and optional values. The help function output for the LMDZ compile script is shown in Figure 2.12.

Only two parameters can be given on the command line as positional parameters: the target and the MPI library version. They can also be changed by editing the script, together with the other parameters for the model which are configurable during compilation (e.g. model specific cpp flags, flags for grid resolution,...).

```

:
Checking the libraries finished
Library update exit status :
ioipsl : 0
psmile.MPI2 : 0
mpp_io : 0
Libraries OK
Now updating : src_parameters ...
gmake[1]: Entering directory `.../SX/build/mod/orchidee_I01.MPI2'
src_parameters is OK
gmake[1]: Leaving directory `.../SX/build/mod/orchidee_I01.MPI2'
Now updating : src_stomate ...
gmake[1]: Entering directory `.../SX/build/mod/orchidee_I01.MPI2'
src_stomate is OK
gmake[1]: Leaving directory `.../SX/build/mod/orchidee_I01.MPI2'
Now updating : src_sechiba ...
gmake[1]: Entering directory `.../SX/build/mod/orchidee_I01.MPI2'
src_sechiba is OK
gmake[1]: Leaving directory `.../SX/build/mod/orchidee_I01.MPI2'
src_parameters : 0
src_stomate : 0
src_sechiba : 0
orchidee ... is up-to-date!
No executable created.
Checking the submodel finished
Submodel update exit status :
src_parameters : 0
src_stomate : 0
src_sechiba : 0
Submodel orchidee OK

```

Figure 2.13: Control messages for submodel compilation output during the LMDZ compilation process. The first lines stem from the library updates launched by submodel ORCHIDEE. The following lines give the control output for the ORCHIDEE source code directories's update. The script always prints the full site dependent path. It is replaced by '...' in the figure.

An extract of the output of the LMDZ compile script check of its libraries and its submodel for the case

that everything is up to date is displayed in Figure 2.13. The extract starts with messages on the exit status of the library compilation (ipsl, psmile, mpp_io). Then, for all submodel (ORCHIDEE) source code subdirectories (parameters, stomate, sechiba), it is checked whether the compilation is up-to-date and the exit status is given.

Structure and assembly of model compile scripts

With each call of `Create_COMP_models.frm` (see Section 2.1.3) a compile script is created for the specific component model, platform, and coupled constellation.

This is done by assembling header files (`*.h`) into a shell script, the compile script, using the `m4` macro processor (Figure 2.14).

The header files contain script fragments specific for a model or a platform or both, or they contain script code that is used for all models on all platforms. The header files with a model name contained in their file name depend on that component model. Those with a node name are specific for that site. Header files without model or node name are included into all compile scripts.

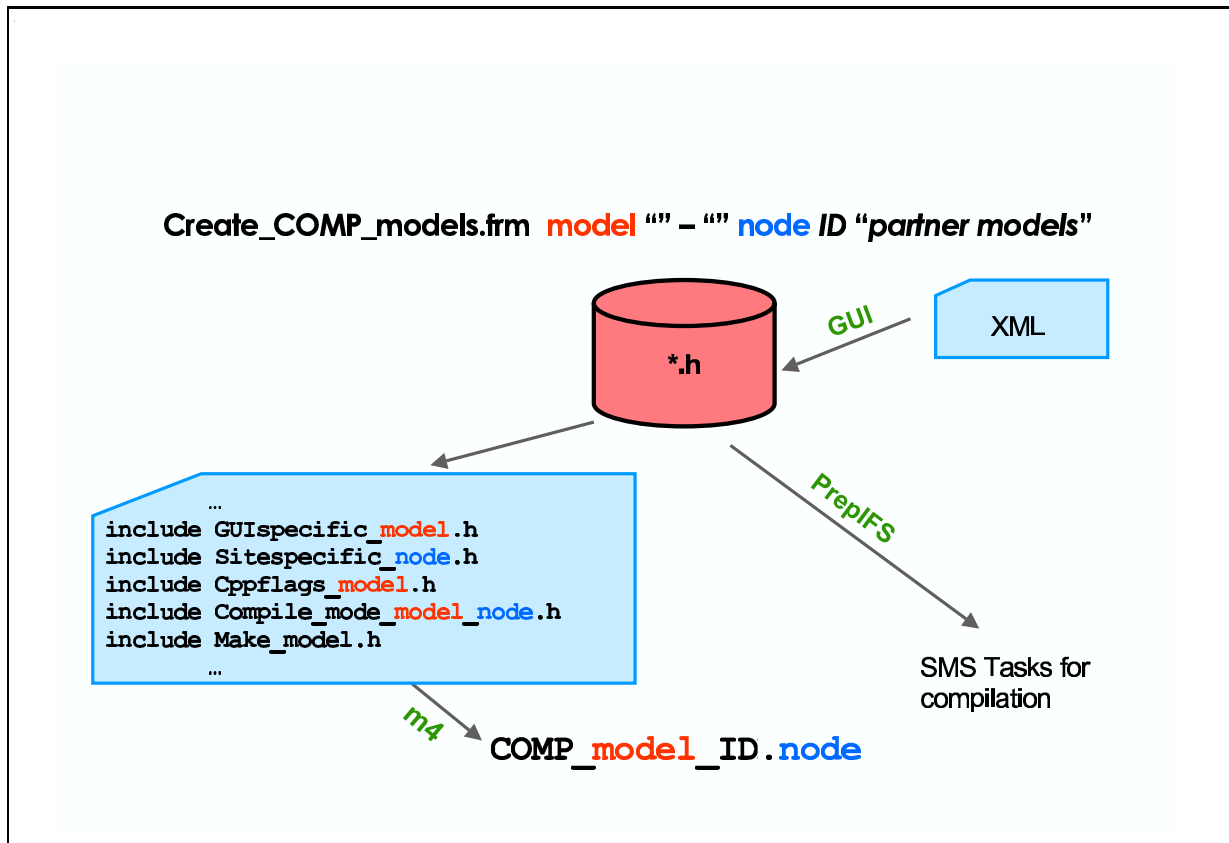


Figure 2.14: The figure gives a graphical presentation of the script assembly process. The left side is for the shell script generation, the right one for the tasks assembly through the GUI system.

Thereby the scripts give a common look&feel for all component models. The maintenance of the system is small, since even for a large number of component models or platforms, there is little redundant or duplicated code (see Section 3.2 of Chapter 3). In order to include a new platform into the system, only the site dependent header files have to be provided. To accommodate a new model, only the header files containing model dependent specifications have to be written (see Section 3.2 of Chapter 3).

Similar to the `cpp` preprocessor, `m4` copies the named header files to the position of the include directives in the `m4` input file. The directive is either `include` or `sinclude`. The first requires the named header file to exist, while files included with the `sinclude` directive are optional. Before include directives are executed, the strings `#{model}` and `#{node}` appearing in the header file names in the

m4 input file are replaced by the model and node name command line parameters of the call of `Create_COMP_models.frm`. The header files are identical to those prepared when the models are configured through the GUI. Exceptions are the files with the `_frm` string in their name which are used by the shell scripts only.

Below, a short description of the contents of the header files is given in the order the include directives appear in the m4 input file. This also describes the structure of the scripts. The complete `Create_COMP_models.frm` script is found in Appendix A. A complete example of a compile script is found in Appendix B.

`Qsub_start_node.h`: At most sites it is possible to compile the model interactively by just running the compile shell script. If a compilation within a queuing system is required the specification for the particular queuing system has to be provided in this file.

`Comments_models.h` and `Comments_models_frm.h`: Comments, including a help function for the usage of the script. The usage, relating to the command lines parameters for the compile script is contained in the 2nd file. It is not used with the GUI system which does not use the parameters in that way.

`Prolog_all.h`: Detection of the compile server's node name and `cd` to the component model directory. Used for all model compile scripts as well as for the library compile script.

`Prolog_models.h`: Detection and echo of the component model name. Used for all model compile scripts.

`Guispecif_all.h`: Specifications needed for all component models and the libraries: MPI version, coupler name, and whether I/O with the `mpp_io` PRISM library will be done.

`Guispecif_models.h`: Specification of a target 'mode': if `forced` is set in this file, a target rebuilt can be enforced.⁹

Dummy specification for the list of partner models. It will be overwritten by parameter 7 given during script generation.

`Guispecif_model.h`: Model dependent default values for the compile mode (see below), the grid acronym (may be empty), the default make target (`all` or `lib`), the version ID (set when the script is created), and a default submodel name (may be empty).

`Guispecif_model_frm.h`: Model dependent configurable specifications not used with the GUI system. Not needed for all models.

`Command_par_models_frm.h`: Definition of help function and overwriting of default values by command parameter values. Not used with the GUI system.

`Input_check_all_frm.h`: Check of message passing and target specification for all model's and the library scripts. Not used with the GUI system.

`Print_par_all.h`: Renaming and export of configurable parameters used for all model's and the library scripts.

`Print_par_models.h`: Renaming and export of configurable parameters used for all models.

`Print_par_model.h`: Check, renaming, and export of model dependent specifications.

`Sitespecific_node.h`: Check of node name and specification of architecture name as well as site dependent path names of libraries and commands.

`Compile_mode_model_node.h`: For each model and each site a couple of different compile modes can be given. The predefined modes are `default`, `debug`, `profile`, and `opt`¹⁰. The compile mode is set in `Guispecif_model.h`.

`OSspecific_node.h`: OS dependent flags of preprocessor, compiler, loader, archive, and assembler etc. commands. Preprocessor flags for configuration of source code for the architecture (see Section 1.2 in Chapter 1).

⁹This is not enabled for all component model makefiles.

¹⁰The system is easily extended by other compile modes with a granularity down to individual users.

`Cppflags_model.h`: Model specific `cpp` flag default values. Flags that depend on parameters specified by the user (through the GUI or in the script) are set accordingly¹¹.

`Cppflags_edit.h`: Concatenation of all `cpp` flags and control output.

`Execname.h`: Built of the executable name. The name depends on the message passing, the component model name, and the version ID. If a submodel is linked to the model, its name also included.

`Libraries_model.h`: The name of this header file is somehow misleading. It contains not only the directory names of the libraries that need to be linked to the model, but also the list of its source code directory names that have to be compiled. The name of the main program has to be entered here as well (if the loader can optionally be called with the script, i.e. if the model has the potential to be a main model).

`Libraries_models.h`: `PSMILE` and `mpp_io` are appended to the individual lists of libraries if needed. The list of libraries of the `tar` target is specified including submodels if any.

`Build_dirs_models.h`: Output of script flow control messages. Making the build directory. Conversion of library path names into loadable library archive names and library prerequisite names.

`Check_libs.h`: Call of the library compile script with the appropriate of libraries set in the script. The exit status of that script is echoed and the run is stopped if not zero.

If the model will be linked to a submodel, the submodel's compile script is called with the appropriate set of parameters as set in the script. The exit status of that script is echoed and the run is stopped if the latter is not zero.

`Add_cond_model.h`: Some frequently changing environment parameters may cause the change of `cpp` flags for some models. This requires the unconditional recompile of binary files which is enforced if this header file is included. It exists for the OGCM MPI-OM only presently.

`Top_makefile_all.h` and `Top_makefile_models.h`: `Top_Makefile_$$` is created. In order to allow parallel compilation of several models with the GUI system, the process ID is appended. The 4 targets described above are defined.¹²

`Make_model.h`: Call of `(g)make` with the input file `Top_Makefile_$$`. `Top_Makefile_$$` is deleted if the target is different from `clean`.

`Status_mods_frm.h`: The exit status for all calls of `(g)make`, i.e. all model source code directories with a makefile is printed. Used with the scripting system only.

`Save_exec.h`: Final modification of the executable name. The executable is moved to `/root/arch/bin`.

`Qsub_end_node.h`: See first item.

¹¹Some naming conventions for `cpp` flags can be found in Chapter 1 Section 1.2.

¹²This is the place where additional optional targets may be included

2.2 Library compilation

The source code of routines of general purpose which have the potential to be used by any model are stored in the library source code branch `/root/src/lib` (see Figure 1.1). As for the component models, each library source code directory with input files for a compiler contains a (low level) makefile.

2.2.1 Low level library makefiles

There are no tools provided to generate the library low level makefiles. They are fully portable, and are downloaded from the PRISM CVS repository with the libraries. The first part of such makefiles containing the general specifications is similar to those of the models. That of the PSMILe library is displayed in Figure 2.15.

```
SRCS1 = $(shell ls ../../../../src/lib/psmile/src/*.F90)
OBJS1 = $(patsubst ../../../../src/lib/psmile/src/%.F90, %.o, $(SRCS1))
SRCS2 = $(shell ls ../../../../src/lib/psmile/src/*.f90)
OBJS2 = $(patsubst ../../../../src/lib/psmile/src/%.f90, %.o, $(SRCS2))
SRCS3 = $(shell ls ../../../../src/lib/psmile/src/*.F)
OBJS3 = $(patsubst ../../../../src/lib/psmile/src/%.F, %.o, $(SRCS3))
VPATH = ./\
        ../../../../src/lib/psmile/src:\
        ../../../../src/lib/psmile/include:\
        ../../../../src/lib/mpp_io/src:\
        ../../../../src/lib/mpp_io/include
LIBRARY = ../../../../lib/libpsmile.${CHAN}.a
clean:
    rm -f i.* *.o *.mod
all:
    $(LIBRARY)
$(LIBRARY): $(OBJS1) $(OBJS2) $(OBJS3)
    $(AR) $(ARFLAGS) $(LIBRARY) $(OBJS1) $(OBJS2) $(OBJS3)
INCLS = -I ../../../../src/lib/psmile/include \
        -I ../../../../src/lib/mpp_io/include
INCLSC = -I ../../../../src/lib/psmile/include \
        -I ../../../../src/lib/mpp_io/include
.SUFFIXES:
.SUFFIXES: .o .f .F .f90 .F90 .c
%.o: %.F90
    $(F90) $(MPPFLAGS) $(F90FLAGS) $(INCLS) -c $<
%.o: %.f90
    $(f90) $(MPPFLAGS) $(f90FLAGS) $(INCLS) -c $<
%.o: %.F
    $(F) $(MPPFLAGS) $(FFLAGS) $(INCLS) -c $<
%.o: %.f
    $(f) $(MPPFLAGS) $(fFLAGS) $(INCLS) -c $<
%.o: %.c
    $(CC) $(CCFLAGS) $(INCLSC) -c $<
```

Figure 2.15: First part of the PSMILe library low level makefile.

The lists of object `*.o` files `OBJS1`, `OBJS2`, `OBJS3` are generated first. These are the prerequisites for the creation of the library archive `$(LIBRARY)`. Note that the library does not contain `*.c` or `*.f` files. The PSMILe library owns an `include` directory which is included in the `VPATH` variable. In addition, it USES FORTRAN90 modules of the `mpp_io` library which therefore also appears in that variable together with its `include` directory.

The PSMILe library is configured for the version of the MPI-[1,2] message passing library which therefore appears in the library archive file name (`$(CHAN)`). This allows to have both versions available for use in the same `/root/` directory.

The makefile contains only the targets `all` and `clean`. With the first target, the default, the library archive is created, with the second target, the build directory of the library is cleaned. The libraries in the `/root/src/lib` have no `lib` target in their makefile.

The rule to make the `$(LIBRARY)` is to include all binary files into the library archive. The OS dependent loader flags are exported from the compile script. The rule for the compilation of the library binaries includes an additional flag `$(MPPFLAGS)` which is not used for model compilation.

For more details refer to Section 2.1.1 where a corresponding makefile of a component model is described.

2.2.2 The library compile script

There is one compile script which is used for the compilation of all libraries in `/root/src/lib` of the PRISM system. It is not downloaded with the source code from the PRISM repository, since it is OS and site dependent, but must be generated for each site by the user.

Create_COMP_libs.frm

The tool to do that is `Create_COMP_libs.frm`. It resides in `/root_/util/compile/frames`.

The help function output of this script is displayed in Figure 2.16. As for the model compile script creation, the device where the standard output and the error output is directed can be specified by the first two parameters. With the third parameter, the default for the model launching option `MPI2` can be changed to `MPI1`. This has impact only for the `PSMILE clim` libraries. The third parameter is the node name which is required only if the compile script is created on a machine other than the compile server.

```

Create_COMP_libs.frm --help
-----
This script may fail if you do not use the right version of m4!
The version you use is : GNU m4 1.4.1
Make sure this is GNU m4 younger than version 1.4!
-----
Usage
-----
$1="/" "-":      directing std output to a file / the screen
$2="/" "-"/"+":  directing err output to a file / the screen / stdout
$3="/" "node":   node name for header files with OS/site specs
$4="/" "MPI1":   message passing
-----

```

Figure 2.16: Help text for the library compile script creation. No parameter input is required.

COMP_libs.node

The library compile script `COMP_libs.node` needs to be created only once for each platform. It is launched by all model compile scripts with the appropriate list of libraries set up in the component model compile script.

All configurable aspects of library compilation can be changed when the script is launched by setting the command line parameters.

One of them, the last of 3, is the message passing MPI library version `MPI1` or `MPI2`. Most component models require this to be `MPI2` which is there the default. It can be changed to `MPI1` when the script is run. This configuration has impact only for the `PSMILE` and `clim` libraries. Depending on the value passed to the script, a library `/root/arch/lib/libpsmile.MP2.a` or `/root/arch/lib/libpsmile.MP1.a` is created.

The second parameter defines the list of libraries which will be updated. The default list consists of the libraries needed by `OASIS3` and the `PSMILE` library, i.e. the libraries of the PRISM coupling software.

The first parameter sets the target. The compile script calls all low level makefiles of all libraries specified in the second parameter with that target if it is `clean` or `all`. With `tar`, a tarfile is prepared with the source code of the specified libraries and the utilities of directory `/root/util`.

The help function output of the script is shown in Figure 2.17 .

```

COMP_libs.ds --help
-----
Usage
-----
$1=make_target      (all/tar/clean; optional)
$2=list_of_libraries (anaig/anaism/clim/etc.; optional)
$3=message_passing  (NONE/MPI1/MPI2; optional)
Default active values are:
make_target        : all
list_of_libraries  : anaig anaism clim fscint mpp_io psmile scrip
message_passing    : MPI2
-----

```

Figure 2.17: Help text for the library compile script. The last lines give the defaults which were set when the script was created.

Structure and assembly of `COMP_libs.node`

As for the component model compile scripts, the library compile script is assembled with the `m4` macro processor by including header files into a frame, the `m4` input file. These header files depend on the site and OS if a node name is part of their file name. They are taken from the same base of header files that is used for the assembly of the component model compile scripts.

Similar to the `cpp` preprocessor, `m4` copies the named header files to the position of the `include` directive in the `m4` input file. The `include` directive requires the named header file to exist, while files included with the `sinclude` directive are optional. Before include directives are executed, the string `#{node}` appearing in the header file names in the `m4` input file are replaced by the node name detected by the system or specified as command line parameters in the call of `Create_COMP_libs.frm`. The complete `Create_COMP_libs.frm` script is found in Appendix D.

The header files are identical to those prepared when the PRISM system is configured through the GUI (see Chapter 4). Exceptions are the files with the `_frm` string in their name which are used by the shell scripts only.

Below, a short description of the content of the header files is given in the order they are included into the compile script. This also describes the structure of the compile script `COMP_libs.node`. The complete `COMP_libs.node` script is found in Appendix E.

`Qsub_start_node.h`.*

`Comments_libs.h` and `Comments_libs_frm.h`: Comments, including a help function for the usage of the script. The usage, relating to the command lines parameters for the script is contained in the 2nd file. It is not used with the GUI system which does not use the parameters in that way.

`Guispecif_all.h`.*

`Guispecif_libs.h`: Default values for the compile mode (`default`, see below), the make target (`all`), and the list of libraries that will be updated ("`anaism anaism clim fscint mpp_io psmile scrip`").

`Command_par_libs_frm.h`: Definition of help function and overwriting of default values by command parameter values. Not used with the GUI system.

`Input_check_all_frm.h`.*

`Input_check_libs_frm.h`: Check of library names in the list of libraries to be updated against a list of valid libraries.

`Print_par_all.h`.*

`Print_par_libs.h`: Renaming of the library list, control output of the action to come, and construction of a list of libraries for use with the `tar` target.

`Prolog_all.h`.*

`Prolog_libs.h`: Detection of the PRISM root source directory and `cd` to that directory.

`Sitespecific_node.h`.*

`Compile_mode_libs_node.h`: Definition of several options for choosing the compiler options (`default`, `debug`, `profile`, and `opt`). In the present release, all libraries are compiled with the same option ¹³.

`OSspecific_node.h`.*

`Cppflags_libs.h`: Library `cpp` flag default values. Flags that depend on parameters specified by the user (through the GUI or in the script) are set accordingly.

`Cppflags_edit.h`.*

*This header file is also used for the assembly of the component model compile scripts. It is described in Section 2.1.3.

¹³This can easily be extended to allow for different versions for each option and each library.

`Build_dirs_libs.h`: Making the build directory for all specified libraries. The makefiles are copied to these directories and the script changes into one of them.

`Top_makefile_all.h`:*

`Top_makefile_libs.h`: The three targets which can be passed to the top level makefile `all`, `clean`, and `tar` are defined.

`Make_libs.h`:

`Make_model.h`: Call of `(g)make` with the input file `../Top_Makefile_$$`. `../Top_Makefile_$$` is deleted afterwards.

`Status_libs_frm.h`: The exit status for all calls of `(g)make` is printed. Used with the scripting system only.

`Qsub_end_node.h`:*

Chapter 3

Extending the SCE

The PRISM SCE is easily extended to accommodate new models and platforms. The extension is facilitated by a highly modularized approach and a common look&feel given by the SCE tools. Using these tools, however, requires that the models meet a number of standards.

If a new component model that is to be adapted to the SCE has already been run successfully on the envisaged platform, the scope of the work is significantly reduced. Otherwise it is recommended to first port the new model in a standalone mode to the platform. Standalone experiments are supported by PRISM. The ocean-only PRISM model MPI-OM can be taken as an example application. Its adaptation to the PRISM system is described in the MPI-M PRISM Model Adaptation Guide by Legutke and Gayler (2004).

The next section describes how a new component model is adapted to the PRISM SCE under the assumption that the platform is already a PRISM compilation site, i.e. that the header files containing the OS and site specifications exist. The extension of the SCE to enable PRISM experiments to be run on a new platform is described in Section 3.2 below.

3.1 Extending the SCE to accommodate a new model

A prerequisite to the use of PRISM tools for compilation is the adaptation of the model's source code to the PRISM source code directory structure.

In addition its source code must meet some essential coding rules. These rules guarantee that the model can be compiled with the compile scripts provided by the system which are based on the GNU (`g`)`make` utilities. These are the requirement that any file containing a FORTRAN90 module is only allowed to contain a single module, and that the file name must be identical to the module name. No two files with the same base name are allowed to exist. Other coding rules are described in the PRISM Software Engineering Process, Coding Rules, and Quality Standard handbook by Mangili et al. (2003).

Figure 3.1 displays how a model source code tree which does not meet the SCE standards can be modified to be PRISM compliant without changing the grouping of routines in different directories. The number of source code directories is not limited. However, they must all be on the same directory tree level (see Section 1.1 of Chapter 1).

If the new component model needs to be linked to a library not yet in the system, the library source code must be stored in the PRISM library source code branch `/root/src/lib`. In the present release of the PRISM software, no library has more than one source code directory, excluding `include` directories with input to the preprocessor only. The library compile script assumes that this directory is called `src`. If that structure can not be used for the new library the PRISM team should be contacted (`prism_help@enes.org`)¹.

¹The script can easily be modified to loop over more than directory and call (`g`)`make` with the makefiles therein.

If the new library has more than one source code directory with input files for a compiler, it should be considered to merge the directories, or to define one library for each directory. The second solution is recommended if there are no interdependencies between the files in the directories in the GNU (g)make sense. The first solution should always be possible since identical file names are a problem anyway and should be eliminated. If there are identical base file name this should also be corrected.

In each of the library source code directories a file called `Makefile` has to be available. Any makefile from one of the existing libraries can be used as template (see Section 2.2.1). Note that the first pairs of lines (Figure 2.15) which create the list of binary prerequisites for the library archive has to appear in the makefile for any valid file name suffix (see Section 1.2) present in the source code directory.

There is no support yet to generate prerequisites for the compiler commands in the library low level makefiles. The list of prerequisites for the binaries must either be taken from an existing script or they may be generated with some preprocessor options (have a look at the `cpp` preprocessor man page). In addition, the new library has to be introduced to the system by including its name into the `Input_check_libs_frm.h` header file of directory `/root/util/compile/frames/include`.

When this is achieved a new library compile script can be generated with the `Create_COMP_libs_frm` tool (see Section 2.2.2). The newly created script is enabled to update the new library. It is recommended to run it directly for the new library only ("`COMP_libs.node all new_libname`"). It is assumed here that the compile server `node` is already a PRISM compile site, i.e. that the OS and site dependent header files are available. Note that once the files for the site specifications exist, they should not be changed by the user without coordination with the PRISM team since they will be used by all other models when they are run at that site.

Problems may occur if the new library can not be compiled with the compiler options used for the other libraries. To fix this the system/PRISM² and/or library administrator should be contacted (see also Section 2.2.2).

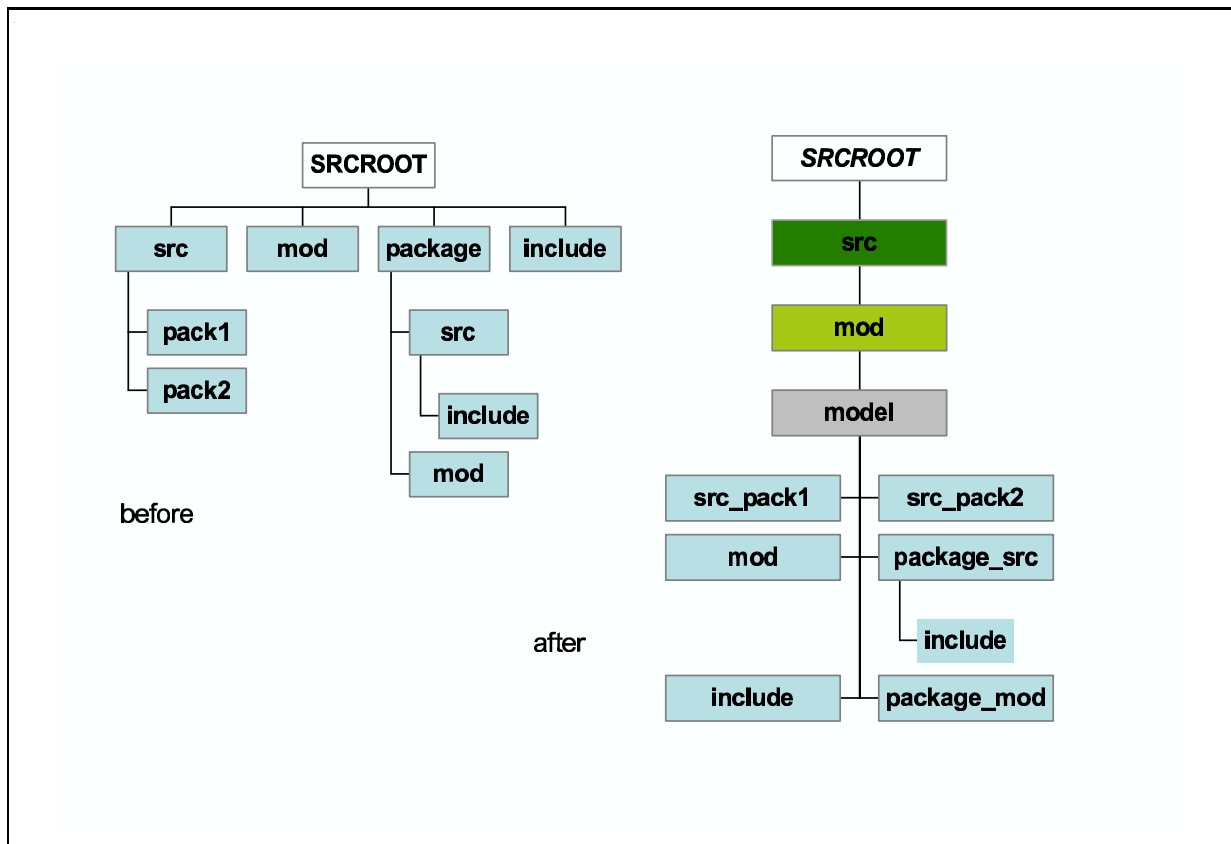


Figure 3.1: Example for a model source code storage adaptation to the SCE

²prism_help@enes.org

Next a `Makefile_1` has to be provided in each source code directory of the new component model containing input for a compiler. Any other `Makefile_1` from any other component model already adapted to the SCE can be used as template (see Section 2.1.1). For each source code directory for which a `Makefile_1` is available the prerequisites for the compiler commands can be generated with `Append_dependencies` (see Section 2.1.2). It gives the source code directories which have to be included into the `VPATH` and `INCLS` variables for the `(g)make` search. If the model has a large number of source code directories with complex interdependencies it may be necessary to increase the number of iterations done by the script to resolve all dependencies. This is done by increasing the value of the `times` variable in the script. If the compilation is not done in one go from scratch, this is an indicator that that variable is too small.

Before the component model compile script can be created, the model has to be introduced to the system by including its name in the list of PRISM models in `Create_COMP_models.frm` contained in the variable `list_of_PRISM_models`.

Interfacing of the component model with the PRISM coupling software, in particular the use of the PSMILe library is not subject of this handbook. This is described in the OASIS3 USER Guide by Valcke et al. (2004a). If an MPI parallel model is interfaced to this software, however, not able to run internally parallelized and without the spawn feature of the MPI-2 message passing library in a coupled mode with OASIS3, the check for an invalid MPI paradigm must be modified accordingly in the same script. Note that the source code of a component model is always the same, independent on its partner models in a coupled constellation, or whether it runs in a standalone mode. Therefore the component models should be configured for the use of the PSMILe library or the exchange with different partner models using conditional compiling controlled by `cpp` flags. Standard names are listed in Section 1.2.

In addition, the model specific header files with the script code fragments have to be newly written for a new model. A list of these files can be obtained from already existing files by typing `'find ./util/compile/frames/include/*/* -name '*_old_model*.h''` where `old_model` is a component model already adapted to the SCE. These files should be copied into new files with the old model's name replaced by the new model's name and the contents have to be modified appropriately.

The compile script can then be created with the script `Create_COMP_models.frm`.

3.2 Extending the SCE to accommodate a new site

To allow a PRISM experiment to be performed on a new site, the SCE and SRE have to be extended by writing a couple of files with the site dependent specifications. Here only the SCE aspects are described. For the extension of the SRE refer to Gayler and Legutke (2004).

The SCE includes a tool to generate compile scripts specific for component models and platforms (Section 2.2.2). The scripts are assembled from a set of files, three of which contain the platform dependent specifications. For a new platform with node name `new_site` these have to be stored in a new directory `include_new_site` which has to be created in `./util/compile/frames`. It must contain the files `OSspecific_new_site.h` and `Sitespecific_new_site.h` with Operating System (OS) dependent parameters and site dependent paths. In addition, one has to write a file `Compile_mode_model_new_site.h` for each component `model` with the correct compiler options. Existing files from a PRISM site can be used as templates.

The compile scripts for the new site are created by typing `Create_COMP_models.frm model "" - "" new_site ID "list of partner models"` in directory `./util/compile/frames`. Running the models on the new site is described in the SRE handbook by Gayler and Legutke (2004). It is recommended to first test the new site dependent features with the TOYCLIM models (refer to Valcke (2004)).

It may happen that a component model contains non-portable source code. New code should be wrapped

into `cpp` flags for conditional compiling. There are default `cpp` flag names for some architectures listed in Section 1.2. It is essential that the new model version can still be run on all old platforms.

Chapter 4

Using the GUI

All PRISM coupled models of the CVS release `prism_2-4` can be configured either as described in this handbook or through the PRISM graphical user interface (GUI) PrePare. The execution of experiments can be scheduled and monitored through the GUI of the SMS(SchedulingMonitoringSupervising) software.

How to install this software and operate the GUIs is described in The PRISM Graphical User Interface (GUI) and Web Services Guide by Constanza et al. (2004). A CD containing the software can be ordered on the PRISM web portal¹. Note that the SMS software is licensed. A license form can be downloaded from the same PRISM web site.

The CD contains the full GUI system. A "Hello world" application gives the user a graphical impression of the functionality of the system without being able to interact.

A second application on the CD runs with the PRISM TOYCLIM coupled toy model. Though there is no 'real' climate model running, the user has the full functionality provided by both the PrePare and SMS software and the PRISM coupling software. All configuration and the running of the 'experiment' is done through the GUIs.

4.1 Configuring compile scripts with PrePare

The configuration of the models at compile time, i.e. of the compile script, is done in the GUI system with a method similar in many aspects to that used on the scripting level. On the scripting level configuration is done by the specific selection of header files and by parameter substitution in these files by the m4 macro processor when the scripts are assembled, and through command line parameters when they are launched. In the GUI system, the scripts are also assembled from header files, however by the SMS software. The header files are mostly identical to those used by m4, differences are related to the usage of the command line parameters in the scripting system (see the list of header files in Section 2.1.3 and their description).

The header files are configured through the GUI by filling in XML files (Figure 2.14) from which they are created. Details about this are found in The PRISM Graphical User Interface (GUI) and Web Services Guide by Constanza et al. (2004).

4.2 Monitoring the compilation with SMS

The compilation process can be monitored through the SMS GUI on a site where it is installed. One option is a local Linux workstation provided the model can be compiled and run there.

During the PRISM project all PRISM models could be configured, compiled, and run through the GUI on the DKRZ machines (Linux, NEC SX-6). However, this site had to be shut down due to security

¹<http://prism.enes.org/Portal>

considerations. It is presently discussed in the PRISM Support Initiative being started up how a new installation can be realized.

Persons with an account on the IBM compute host at the ECMWF site have the option to use the GUIs there through the web. However, only the TOYCLIM models had been ported to that site at project end.

Bibliography

- Guilyardi, E., R. Budich, G. Brasseur, G. Komen, 2002: PRISM System Specification Handbook V.1. PRISM Report Series No 1. , 239 pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report1_pdf).
- Valcke, S., A. Caubel, R. Vogelsang, and D. Declat, 2004a: OASIS3 User Guide (oasis3_prism_2-4). PRISM Report Series, No 2. , 5th Ed., nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report2_pdf).
- Valcke, S., R. Redler, R. Vogelsang, D. Declat, H. Ritzdorf, T. Schoenemeyer, 2004b: OASIS4 User Guide. PRISM Report Series No 3. , nn pp.,
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report3_pdf).
- Legutke, S. and V.Gayler, 2004: The PRISM Standard Compilation Environment, PRISM Report Series No 4. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report4_pdf).
- Gayler, V. and S. Legutke, 2004: The PRISM Standard Running Environment, PRISM Report Series, No 5. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report5_pdf).
- Constanza, P., C. Larsson, C. Linstead, X. Le Paster, and N. Wedi, 2004: The PRISM Graphical User Interface (GUI) and Web Services Guide, PRISM Report Series, No 6. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report6_pdf).
- Valcke, S., 2004: The TOYCLIM PRISM Coupled Model Adaptation Guide, PRISM Report Series, No 7. , 70 pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report7_pdf).
- Legutke, S. and V.Gayler, 2004: The MPI-M PRISM Earth System Model Adaptation Guide, PRISM Report Series No 8, nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report8_pdf).
- Demory, M.-E., 2004: THE IPSL_CM4 coupled model adaptation guide, PRISM Report Series No 9. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report9_pdf).
- Planton, S., xxxx, 2004: Meteo France PRISM Model Adaptation, PRISM Report Series No 10. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report10_pdf).
- Kastowski, M., xxxx 2004: MPI Jena PRISM Model Adaptation, PRISM Report Series No 11. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report11_pdf).
- Meier-Fleischer, K., xxxx 2004: Low end visualization of PRISM model output, PRISM Report Series No 12. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report12_pdf).
- Carril, A., R. Budich, S. Valcke, P. van Velthoven, S. Legutke, T. Schoenemeyer, 2004: The wp5 TOY-CLIM demo run report, PRISM Report Series, No 13. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report13_pdf).
- Carril, A., xxxx 2004: The PRISM Demonstration Runs, PRISM Report Series, No 14. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report14_pdf).

-
- Carter, M., xxx 2004: The PRISM Data Processing and Visualisation System, PRISM Report Series No 15. , nn pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report15_pdf).
- Mangili, A., M. Ballabio, M. Djordje, L. Kornblueh, R. Vogelsang, and P. Bourcier, 2003: PRISM Software Engineering Process, Coding Rules, and Quality Standard, PRISM Report Series No 16. , 31 pp.
(http://prism.enes.org/Results/Documents/PRISM_Reports/Report16_pdf).

Appendix A

Component model compile script generation tool

Script for the creation of model compile scripts

```
#!/bin/ksh
#####
#
#   Script to generate a component model compile script
#
#   File       : ~/util/compile/frames/Create_COMP_models.frm
#
#   Stephanie Legutke, MPI-HH, M&D           Dec 14, 2004
#
#   This script is used to generate a compile script for a PRISM component model.
#   The script is specifically created for the target platform.
#   Depending on the node name it selects site and OS specifics.
#   The node name is either detected with 'uname -n' or, if the script is not
#   created on the target machine, it must be specified by the user.
#   The generated compile script is moved to ~/src/mod/'modelname'
#   after successful generation.
#   The script uses m4 for preprocessing. It has been tested with
#   GNU m4 version 1.4. Other, older or non-GNU versions of m4 may fail!
#
#   This script is called interactively only.
#
#   7 positional parameters :
#
#   Usage : Create_COMP_models.frm \
#           'model_name' \
#           [message_passing [std out [std err [node[version[partner models]]]]]]
#
#   $1='model_name'           (required)
#   $2="/"MPI1"/"NONE"       message passing
#   $3="/"-"                 directing std output to a file / the screen
#   $4="/"-"+"               directing err output to a file / the screen / stdout
#   $5="/"'node'"           node name for header files with OS/site specs
#   $6="/"'version'"        version acronym for differentiation of executables
#   $7="/"partner models"   models of the coupled model constellation
#
#   If not called with all parameters, the defaults are:
#   $2 MPI2
#   $3 gmake directs stdout to a file
#   $4 gmake directs stderr to a file
#   $5 node name 'uname -n' of the machine where the script is run
#   $6 D10
#   $7 " " (standalone)
#
#   Note: if the model has not yet been ported to a machine (specified by
#   node name) a message like: infile.m4:53: m4: Cannot open include/...
#   for some model or machine dependent header file will be given.
#   These files have to be provided then.
#####
set -x
if [ "$1" = "" ] || [ "$1" = "--" ] || [ "$1" = "help" ] || [ "$1" = "--help" ]; then
echo
echo '-----'
echo ' '
echo ' This script may fail if you do not use the right version of m4!'
echo ' The version you use is : ' `m4 --version`
echo ' Make sure this is GNU m4 younger than version 1.4!'
echo ' '
echo '-----'
echo ' Usage -----'
echo ' $1=model name                               (none) '
echo ' $2="/"MPI1"/"NONE": message passing         (MPI2) '

```

```

echo ' $3="/"-"-": std output of compile script -> file / screen          (file)'
echo ' $4="/"-"-"/+": err output -> file / the screen / stdout          (file)'
echo ' $5="/"-"node name": node name of compile server                ('uname -n')'
echo ' $6="version acronym"                                           (none)'
echo ' $7="list of participating models"                               (none)'
echo ''
echo ' Specification of defaults is not allowed: write e.g.'
echo '         Create_COMP_models.frm oasis3 "" - "" ""'
echo ' instead of'
echo '         Create_COMP_models.frm oasis3 "MPI2" - "" ""'
echo '-----'
exit
fi
##
## Model names for header file selection
##
list_of_PRISM_models="arpege_climat4 echam5 hamocc ham lim lmdz mozart mpi-om \
oasis3 opa orchidee pisces toy4opa toyatm toyche toyoce toy4arpege"
valid_model=no
for spec_model in ${list_of_PRISM_models}; do
  if [ "$1" = "$spec_model" ]; then
    valid_model=yes
    model=$1
  fi
done
if [ "${valid_model}" = "no" ]; then
  echo ' The model name is not a valid PRISM model: '$1
  echo ' The script is stopped!'
  exit 1
fi
##
## Model version
##
if [ "$6" = "" ]; then
  version=D10
else
  version=$6
  echo "*"
  echo "*" Compile script for model $model, version=$version is created"
fi
##
## If no partner models are specified with cpl_to the standalone version is taken as default.
## This may not be possible for all models.
##
if [ "$7" = "" ] || [ "$7" = " " ]; then
  cpl_to=" "
else
  for modell in $7; do
    valid_model=no
    for model2 in ${list_of_PRISM_models}; do
      if [ "$modell" = "$model2" ]; then
        valid_model=yes
      fi
    done
    if [ "${valid_model}" = "no" ]; then
      echo ' One of the models names is not a valid PRISM model: '$modell
      echo ' The script is stopped!'
      exit 1
    fi
  done
  if [ "${valid_model}" = "yes" ]; then
    cpl_to=$7
    echo "*"
    echo "*" Partner component models are: ${cpl_to}"
  fi
fi
##
## HOST, OS, node name for header selection
##
echo ''
node='uname -n'
build='uname -s'
if [ "$5" != "" ] && [ "$5" != $node ]; then
  echo ''
  echo Warning: the node specified is not the node of this machine.
  echo Ensure that the site/OS specific header files exist.
  node=$5
  build=""
  echo '* Creating compile script'
  echo '         for '$model'.'
else
  echo '* Creating compile script'
  echo '         on/for a '$build' platform with node name '$node','
  echo '         for '$model'.'
fi
nodel_4='echo ${node} | cut -c1-4'
if [ ${nodel_4} = xbar ]; then node=${nodel_4}; fi
if [ ${nodel_4} = hpca ]; then node=${nodel_4}; fi
nodel_2='echo ${node} | cut -c1-2'
if [ ${nodel_2} = ds ]; then node=${nodel_2}; fi
nodel_6='echo ${node} | cut -c1-6'
if [ ${nodel_6} = total1 ]; then node=${nodel_6}; fi
echo '* Abbreviated node name is '$node'.'
##
## Checking consistency of message passing parameter and model
##
mespas=irgendwas
messpass=MPI2

```

```

if [ "$2" != "" ]; then
if [ "$2" = "MPI1" ] || [ "$2" = "NONE" ]; then
  if [ "$2" = "MPI1" ]; then
    if [ "$model" = "echam5" ] ||
      [ "$model" = "mpi-om" ] ||
      [ "$model" = "hamocc" ] ||
      [ "$model" = "pisces" ] ||
      [ "$model" = "psoce" ] ; then
      echo 'MPI1 for '$model' is not possible.'
      exit 1
    else
      messpass=$2
      mespas=MPI2
    fi
  fi
  if [ "$2" = "NONE" ]; then
    if [ "$model" = "echam5" ] ||
      [ "$model" = "" ] ; then
      echo 'NONE for '$model' is not possible.'
      exit 1
    else
      messpass=$2
      mespas=MPI2
    fi
  fi
else
  echo 'ERROR:This message passing may not be specified:'$2
  exit 1
fi
fi
echo '* The message passing will be '$messpass'.'
#
# Redirecting standard output and err output
#
outout='stdout=$SCRIPTDIR/COMP_${MODEL_DIR}.log'
if [ "$3" != "" ]; then
  if [ "$3" = "-" ]; then
    outout='stdout='tty''
    echo '* (g)make standard output will directed to screen.'
  else
    echo 'This values is not allowed for parameter 3 (std out)'
    exit
  fi
else
  echo '* (g)make standard output will be directed to a file.'
fi
errout='stderr=$SCRIPTDIR/COMP_${MODEL_DIR}.err'
if [ "$4" != "" ]; then
  if [ "$4" = "-" ]; then
    echo '* (g)make standard error will directed to screen.'
    errout='stderr='tty''
  elif [ "$4" = "+" ]; then
    echo '* (g)make standard error will directed to stdout.'
    errout='stderr="$stdout"'
  else
    echo 'This values is not allowed for parameter 4 (std err)'
    exit
  fi
else
  echo '* (g)make standard error will be directed to a file.'
fi
#
# Change to directory of script
#
scriptdir='dirname $0'
cd $scriptdir
# /u/fj/mfuj/mfuj004/bin/m4" ; xbar
cat > infile.m4 <<EOF
#!/bin/ksh
changequote([,])dnl
changeocom
dnl
dnl Embedding as batch (not needed on all nodes)
dnl
define(comp_Model,comp_${model})dnl
sinclude(include_${node}/Qsub_start_${node}.h)dnl
undefine([comp_Model])dnl
dnl
dnl Comments, usage, history
dnl
define(comp_Model,${model})dnl
define(COMP_comp_Model,COMP_${model})dnl
include(include/Comments_models.h)dnl
include(include/Comments_models_frm.h)dnl
undefine([comp_Model])dnl
undefine([COMP_comp_Model])dnl
scriptdir='\dirname \"$0\''
cd \"$scriptdir
export SCRIPTDIR='\pwd\''
node='\uname -n\''
dnl
dnl HOST, OS, node name ( <libprolog_oasis3.h>)
dnl
include(include/Prolog_all.h)dnl
include(include/Prolog_models.h)dnl
export TARFILE='${MODEL_DIR}_\date +%y%m%d\'.tar

```

```

dnl
dnl  GUI input for all compile scripts
dnl
changeocom(#)
define(${mespas},${messpass})dnl
include(include/Guispecif_all.h)dnl
undefine([${mespas}])dnl
changeocom
dnl
dnl  GUI input for all model compile scripts
dnl
changeocom(#)
define(Invalid_model,"${cpl_to}")dnl
include(include/Guispecif_models.h)dnl
undefine([Invalid_model])dnl
changeocom
dnl
dnl  GUI input depending on model
dnl
changeocom(#)
define(_ModVers,_${version})dnl
include(include/Guispecif_${model}.h)dnl
undefine([_ModVers])dnl
sinclude(include/Guispecif_${model}_frm.h)dnl
changeocom
dnl
dnl  Command parameter input (scripting only)
dnl
include(include/Command_par_models_frm.h)dnl
include(include/Input_check_all_frm.h)dnl
dnl
dnl  Parameter renaming and printing
dnl
include(include/Print_par_all.h)dnl
include(include/Print_par_models.h)dnl
include(include/Print_par_${model}.h)dnl
dnl
dnl  Site and OS specifics
dnl
include(include_${node}/Sitespecific_${node}.h)dnl
include(include_${node}/Compile_mode_${model}_${node}.h)dnl
include(include_${node}/OSSpecific_${node}.h)dnl
dnl
dnl  Non Site and OS dependent cpp flags
dnl
include(include/Cppflags_${model}.h)dnl
include(include/Cppflags_edit.h)dnl
dnl
dnl  Name of executable
dnl
include(include/Execname.h)dnl
dnl
dnl  Libraries and directories
dnl
include(include/Libraries_${model}.h)dnl
include(include/Libraries_models.h)dnl
include(include/Build_dirs_models.h)dnl
dnl
dnl  Check library update status : scripting only
dnl
include(include/Check_libs.h)dnl
dnl
dnl  Additional update conditions
dnl
sinclude(include/Add_cond_${model}.h)dnl
dnl
dnl  Create top level Makefile
dnl
include(include/Top_makefile_all.h)dnl
define(NodeName,$node)dnl
include(include/Top_makefile_models.h)dnl
undefine(NodeName)dnl
define(outdev,${outout})dnl
outdev
undefine([outdev])dnl
define(errdev,${errout})dnl
errdev
undefine([errdev])dnl
if [ -f \${BLDROOT}/${model}.status ]; then rm \${BLDROOT}/${model}.status; fi
dnl
dnl  Make model
dnl
include(include/Make_model.h)dnl
dnl
dnl  Print update status : scripting only
dnl
include(include/Status_mods_frm.h)dnl
dnl
dnl  Save/rename the executable
dnl
include(include/Save_exec.h)dnl
dnl
dnl  Embedding as batch (not needed on all nodes)
dnl
sinclude(include_${node}/Qsub_end_${node}.h)dnl
exit

```

```
m4exit
EOF
#
# Run m4.
#
m4 infile.m4 > COMP_${model}
status=$?
if [ $status -eq 0 ]; then
  if [ ! ${model} = "oasis3" ]; then
    Compile_script=COMP_${model}_${node}
  else
    Compile_script=COMP_${model}_${messpass}.${node}
  fi
  mv COMP_${model} ../../../../src/mod/${model}/${Compile_script}
  chmod 755 ../../../../src/mod/${model}/${Compile_script}
  olddir=`pwd`
  cd ../../../../src/mod/${model}
  moddir=`pwd`
  echo "* The model directory is " $moddir
  echo "* The compile script name is "$moddir/${Compile_script}
  echo ' '
else
  echo "An error occurred! Status="$status
  echo ' '
fi
cd $olddir
rm infile.m4
exit
```

Appendix B

Component model compile script

Compile script for LMDZ

(version ID I02 on a SX NEC machine with node name ds)

```
#!/bin/ksh
#####
#
#   C O M P I L E - script for maintaining PRISM libraries.
#
#   File       : ~/util/COMP_libs
#
#   This script is used to compile or clean any PRISM library.
#   This include libraries used exclusively in any of the models
#   or those used by the coupler only, or those shared by the models.
#
#   When needed, different versions of the libraries are created.
#   At the moment these are MPI1/MPI2 libraries.
#
#   This script is called directly by the administrator or SMS
#   or by the model's compile scripts (on scripting level).
#   When called by the model's compile scripts it checks and updates
#   the libraries needed to compile the models and only those.
#
#####
#
#   4 positional parameters are possible:
#   $1=make_target      (optional)
#   $2=list_of_libraries (optional)
#   $3=message_passing  (optional)
#   $4=calling_model    (optional)
#
#   Usage : COMP_libs      ["make_target" ["list_of_libraries" \
#                           ["message_passing" \
#                           ["calling_model"]]]]
#
#   If called with no parameters, the defaults given below
#   are used. If parameter n is specified; the parameter 1,...n-1
#   must be specified as well.
#   Note: parameter 4 is not used on the scripting level.
#
#####
set -e
##### Start of user specifications (defaults) #####
#
# message_passing          : NONE / MPI 2/1
message_passing=MPI2
# coupler                  : oasis3
# use_key_noIO             : whether mpp_io will be used for I/O
#
coupler=oasis3
use_key_noIO=no
# list_of_libraries       : any list of PRISM libraries possible
# compile mode            : default/opt/..
compile_mode=default
use_NetCDF=yes
list_of_libraries=" anaosg anaism clim fscint mpp_io psmile scrip"
# make_target             : clean/all/tar/lib
make_target=all
##### Command line parameter #####
#
# Help function
#
if [ "$1" = "--" ] || [ "$1" = "help" ] || [ "$1" = "--help" ] ; then
  echo '----- Usage -----'
  echo ' $1=make_target      (all/tar/clean; optional)'
  echo ' $2=list_of_libraries (anaosg/anaism/clim/etc.; optional)'

```

```

echo ' $3=message_passing (NONE/MPI1/MPI2; optional)'
echo ' '
echo ' Default active values are:'
echo ' make_target      : '${make_target}'
echo ' list_of_libraries : '${list_of_libraries}'
echo ' message_passing   : '${message_passing}'
echo ' -----'
exit 1
fi
# Overwrite with command line parameter if specified
[ "$1" = "" ] || make_target=$1
[ "$3" = "" ] || message_passing=$3
#
# if no message passing is specified, only compile the interpolation libraries
#
if [ "${message_passing}" = "NONE" ]; then
  list_of_libraries=" anaimg anaism fscint scrip"
fi
[ "$2" = "" ] || list_of_libraries=$2
set -u
##### Input checks : all #####
#
if [ "${make_target}" != "clean" ] && \
  [ "${make_target}" != "all" ] && \
  [ "${make_target}" != "lib" ] && \
  [ "${make_target}" != "tar" ]; then
  echo " Invalid target : "${make_target}"
  echo " The task is stopped!"
  exit 1
fi
if [ "${message_passing}" != "MPI2" ] && \
  [ "${message_passing}" != "MPI1" ] && \
  [ "${message_passing}" != "NONE" ]; then
  echo " Invalid option for message passing : "${message_passing}"
  echo " The task is stopped!"
  exit 1
fi
##### Input checks : libraries only #####
#
for library in ${list_of_libraries}; do
  if [ "${library}" != "anaimg" ] && \
    [ "${library}" != "anaism" ] && \
    [ "${library}" != "blas" ] && \
    [ "${library}" != "clim" ] && \
    [ "${library}" != "fscint" ] && \
    [ "${library}" != "ioips1" ] && \
    [ "${library}" != "lapack" ] && \
    [ "${library}" != "linpack" ] && \
    [ "${library}" != "mpp_io" ] && \
    [ "${library}" != "netcdf90" ] && \
    [ "${library}" != "psmile" ] && \
    [ "${library}" != "scrip" ] && \
    [ "${library}" != "libV4" ] && \
    [ "${library}" != "svipc" ] && \
    [ "${library}" != "support" ]; then
    echo " Invalid library : "${library}"
    echo " The task is stopped!"
    exit 1
  fi
done
##### Set and print input parameter: all #####
#
export COUPLER=${coupler}
export MAKETARGET=${make_target}
echo " "
echo "Make target : "${MAKETARGET}
export CHAN=${message_passing}
##### Set and print input parameters: libs #####
#
export srclibs=${list_of_libraries}
echo " "
if [ "${MAKETARGET}" = "all" ]; then
  echo "Libraries which will be updated : "
  echo $srclibs
elif [ "${MAKETARGET}" = "clean" ]; then
  echo "Libraries which will be cleaned : "
  echo $srclibs
elif [ "${MAKETARGET}" = "tar" ]; then
  echo "A tar-file of libraries will be created with:"
  echo $srclibs
fi
#####
#
# Libraries for all models
#
TARFILE_DIRS=""
for srclib in $srclibs; do
  TARFILE_DIRS=${TARFILE_DIRS} src/lib/${srclib}
done
export TARFILE_DIRS
scriptdir=`dirname $0`
cd $scriptdir
export SCRIPTDIR=`pwd`
node=`uname -n`

```

```

##### Prolog #####
#
# 'Prolog' : Node name and operating system etc.
#
export NODE=$node
echo ' '
echo 'This script runs on node ' $NODE'.'
cd $SCRIPTDIR
cd ..
export SRCROOT=`pwd`
# echo 'PRISM top source root directory is '$SRCROOT
export TARFILE=libs_`date +%y%m%d`.tar
##### Site and OS dependent specifications #####
#
#           ds      : part of name of header file
#           NODE    : where the model is      compiled
export NODE=`echo ${NODE} | cut -c1-2`
if [ ${NODE} != ds ]; then
    echo ' The OS/Site dependent specifications are not for this machine.'
    echo ' Node name of this machine:' ${NODE}
    echo ' Script setup is for use on a ds node (cross compiling for SX)'
if [ "$MAKETARGET" != "tar" ]; then
    echo ' The task is stopped!'
    exit 1
fi
fi
if [ "$MAKETARGET" = "lib" ]; then
    echo ' '
    echo 'No executable will be created. '
fi
export ARCH=SX
export BLDROOT=$SRCROOT/$ARCH
set +u
if [[ -z "$LD_LIBRARY_PATH0" ]]; then
    LD_LIBRARY_PATH0=""
fi
set -u
. /SX/opt/etc/initsx.sh
export MPIROOT=${SX_BASE_MPI}
export MPI_LIB="-L${SX_BASE_MPI}/lib0 -lmpi"
export MPI_LIB=""
export MPI_INCLUDE=${SX_BASE_MPI}/include
export NETCDFROOT=/pool/SX-6/netcdf/netcdf-3.5.0
export NETCDF_LIB="-L${NETCDFROOT}/lib -lnetcdf_c059_f285"
export NETCDF_INCLUDE=${NETCDFROOT}/include
export LIB1="-L/SX/opt/MathKeisan/inst/lib -llapack -lblas -llinpack"
export LIB2="-L/pool/SX-6/NAG/fnne504db -lnag "
export LIB3=" "
export SYS_INCLUDE=${SX_BASE_CPLUS}/include/C++
export SYS_INCLUDE=./
if [ "$MAKETARGET" = "all" ] || [ "$MAKETARGET" = "lib" ]; then
    echo ' '
    echo 'Compiler revision : '
    echo ` ${SX_BASE_F90}/bin/sxf90 -V `
fi
export F90com=${SX_BASE_MPI}/bin/sxmpif90
export f90com=${F90com}
export Fcom=${F90com}
export fcom=${F90com}
export cc=${SX_BASE_MPI}/bin/sxmpic++
export ar=${SX_BASE_CROSS}/bin/sxar"
export as="sxas"
export cp=cp
PATH=$SRCROOT/util:$PATH
alias make="gmake"
# Compile modes (specified by user): libraries on a ds* node (cross/cumulus)
#
# Note : the libs must be searched cyclically now for oasis3!
export OPENMP=no
if [ $compile_mode = default ]; then
    optc=" "
    optl="-Wl,-h lib_cyclic"
    optf=" -Wf,-pvctl noassume loopcnt=5000000 "
elif [ $compile_mode = debug ]; then
    echo "not yet possible"
    exit 1
elif [ $compile_mode = profile ]; then
    echo "not yet possible"
    exit 1
elif [ $compile_mode = opt ]; then
    optc="-Chopt"
    optf="-Chopt"
    optl="-Wl,-Z 1000000 -pi line=1000"
elif [ $compile_mode = ad_hoc ]; then
    echo "with this option the options are set through GUI input"
    exit 1
else
    echo "Invalid compile mode"
    exit 1
fi
#
# All models should use cpp flag __SX for activation of platform dependent
# source code on SUPER-SX machines !
#

```

```

# `uname -n` : ds (cross compiling only); cs17
# `uname -s` : SUPER-UX
# `uname -m` : SX-6
#
# With old compiler revision which do not allow for the
# option  optdbl="-Wf,-A idbl4" use:
#         optdbl="-Wf,-A db14" AND -D__SXdbl4
#
export CPPFLAG="-D__SX -D__SX__ "
export CPPFLAGF90start=""
export CPPFLAGSEP=" "
optdbl="-Wf,-A idbl4"
lstflags="-Wf,-L fntlist transform"
if [ "$OPENMP" = "yes" ]; then
    Popt=openmp
    CPPFLAG=${CPPFLAG}" -D__openmp -DUSE_OMP"
else
    Popt="stack"
fi
export F90FLAG="-EP -P$Popt ${optf} ${optl} ${optdbl} ${lstflags} "
export FFLAG=${F90FLAG}
export f90FLAG=${F90FLAG}
export fFLAG=${F90FLAG}
export LDFLAG="-P$Popt ${optf} ${optdbl}          ${optl} "
export CCFLAG=${optc}
export AFLAG="-h float0 -m"
export AFLAG="-float0 -e b" #pw
export AFLAG="-h sx6 -m"
export ARFLAG="ruv"
export MPPFLAG=" "
export I4mods="I"
#####
# Libraries CPP flags not depending on site or OS
#
CPPFLAG=${CPPFLAG} -Duse_comm_${CHAN} -Duse_libMPI"
if [ ${use_key_noIO} = yes ]; then
    CPPFLAG=${CPPFLAG} -Dkey_noIO"
fi
if [ ${use_NetCDF} = yes ]; then
    CPPFLAG=${CPPFLAG} -Duse_netCDF"
fi
CPPFLAG=${CPPFLAG} -DNC_DOUBLE"
export CPPFLAGCC=${CPPFLAG}
CPPFLAGF90=""
for flag in $CPPFLAG ; do
    CPPFLAGF90=${CPPFLAGF90}${CPPFLAGSEP}$flag
done
if [ "$CPPFLAGF90" != "" ]; then
    CPPFLAGF90=${CPPFLAGF90start}${CPPFLAGF90}
fi
export CPPFLAGF90
if [ "$MAKETARGET" = "all" ] || [ "$MAKETARGET" = "lib" ]; then
    echo '
    echo "CPP flags used with the cc command: "
    echo $CPPFLAGCC
    echo '
    echo "CPP flags used with the f90 command: "
    echo $CPPFLAGF90
fi
##### Directories #####
# Container 'Build the build' : make build directories if needed
#
libs=${BLDROOT}/lib
[ -d $libs ] || mkdir -p $libs
lib=${BLDROOT}/build/lib
BLDLIBS=" "
bldlib=""
for library in $srclibs; do
    if [ $library = psmile ] || \
       [ $library = clim ] ; then
        bldlib=${library}.${CHAN}
    else
        bldlib=${library}
    fi
    if [ ! -d ${lib}/${bldlib} ]; then
        echo 'Making build directory '${lib}/${bldlib}' ...'
        mkdir -p ${lib}/${bldlib}
    fi
    ${cp} -p ${SRCROOT}/src/lib/$library/src/Makefile ${lib}/${bldlib}
    BLDLIBS=${BLDLIBS}" "${bldlib}
done
export BLDLIBS
cd ${lib}/${bldlib}
#####
# Create the top level Makefile
#
cat > Top_Makefile_$$ <<'EOF1'
export
SHELL = /bin/ksh
builddir = ./
prefix = ../../../../

```

```

includedir = ${prefix}/include
configdir = ${prefix}/config
exec_prefix = ../../..
top_builddir = $(exec_prefix)/build
bindir = $(exec_prefix)/bin
libdir = $(exec_prefix)/lib
INCLUDES = -I$(NETCDF_INCLUDE) -I$(MPI_INCLUDE) -I$(SYS_INCLUDE)
F90       = ${F90com}
f90       = ${f90com}
F         = ${Fcom}
f         = ${fcom}
CC        = ${cc}
CPP       = ${cc} -E
AR        = ${ar}
AS        = ${as}
AFLAGS    = $(AFLAG)
ARFLAGS   = $(ARFLAG)
CCFLAGS   = -I$(configdir) -I$(SYS_INCLUDE) $(CCFLAG) $(CPPFLAGCC)
F90FLAGS  = $(F90FLAG) $(INCLUDES) $(CPPFLAGF90)
FFLAGS    = $(FFLAG) $(INCLUDES) $(CPPFLAGF90)
f90FLAGS  = $(f90FLAG) $(INCLUDES) $(CPPFLAGF90)
fFLAGS    = $(fFLAG) $(INCLUDES) $(CPPFLAGF90)
COUPLE    = ${COUPLER}
MPPFLAGS  = $(MPPFLAG)
LIBDIRS   = ${BLDLIBS}
all:
@for DIR in $(LIBDIRS) ;\
do \
    back=`pwd` ; \
    echo ` ` ; \
    cd ../../lib/$$DIR ; \
    $(MAKE) -f Makefile $(MAKETARGET); status=$$? ; \
    echo $$DIR `:` $$status >> $(BLDROOT)/lib.status ; \
    if [ $$status != 0 ] ; then \
        echo "Exit status from make was $$status" ; \
    fi ; \
    cd $$back ; \
done
clean:
@for DIR in $(LIBDIRS) ;\
do \
    back=`pwd` ; \
    echo ` ` ; \
    cd ../../lib/$$DIR ; \
    $(MAKE) -f Makefile $(MAKETARGET); status=$$? ; \
    echo $$DIR `:` $$status >> $(BLDROOT)/lib.status ; \
    if [ $$status != 0 ] ; then \
        echo "Exit status from make was $$status" ; \
    fi ; \
    cd $$back ; \
done
tar:
cd $(prefix) ; \
tar -cvf ${TARFILE} \
${TARFILE_DIRS} \
util/compile
EOF1
# util/COMP_libs.ds \
stdout=$SCRIPTDIR/COMP_libs.log
stderr=$SCRIPTDIR/COMP_libs.err
if [ -f $BLDROOT/lib.status ] ; then rm $BLDROOT/lib.status; fi
#####
# Execute makefile : -p : listing of all default rules
#                  -d : debug
#                  -k : dont stop when error occurs
#
mv Top_Makefile_$$ ../Top_Makefile_$$
if [ -f ${stdout} ] ; then
    rm -f ${stdout}
fi
if [ -f ${stderr} ] ; then
    rm -f ${stderr}
fi
make -f ../Top_Makefile_$$ $MAKETARGET -k 1>${stdout} 2>${stderr}
rm ../Top_Makefile_$$
#
# Print compilation status
#
if [ $MAKETARGET != "tar" ] ; then
    echo ` `
    cat $BLDROOT/lib.status
    set +e
    grep -v `:` 0' $BLDROOT/lib.status
    if [ $? != 0 ] ; then
        echo 'Libraries ...'
        echo $srclibs
        if [ $MAKETARGET = all ] ; then echo '... are up-to-date!' ; fi
        if [ $MAKETARGET = clean ] ; then echo '... are cleaned!' ; fi
        echo ` `
    else
        if [ $MAKETARGET = all ] ; then echo 'Updating libraries '$srclibs' failed !' ; fi
    fi
fi

```

```
    if [ $MAKETARGET = clean ]; then echo 'Cleaning libraries '$srclibs' failed !'; fi
    echo 'Compilation is stopped!'
    exit 1
fi
set -e
fi
echo ' '
#
#####
exit
```

Appendix C

Example library makefile

Makefile for the PSMILE library (first part only)

```
SRCS1 = $(shell ls ../../../../src/lib/psmile/src/*.F90)
OBJS1 = $(patsubst ../../../../src/lib/psmile/src/%.F90, %.o, $(SRCS1))
SRCS2 = $(shell ls ../../../../src/lib/psmile/src/*.f90)
OBJS2 = $(patsubst ../../../../src/lib/psmile/src/%.f90, %.o, $(SRCS2))
SRCS3 = $(shell ls ../../../../src/lib/psmile/src/*.F)
OBJS3 = $(patsubst ../../../../src/lib/psmile/src/%.F, %.o, $(SRCS3))
VPATH = ./:\
        ../../../../src/lib/psmile/src:\
        ../../../../src/lib/psmile/include:\
        ../../../../src/lib/mpp_io/src:\
        ../../../../src/lib/mpp_io/include
LIBRARY = ../../../../lib/libpsmile.${CHAN}.a
clean:
    rm -f i.* *.o *.mod
all:
    $(LIBRARY)
$(LIBRARY): $(OBJS1) $(OBJS2) $(OBJS3)
    $(AR) $(ARFLAGS) $(LIBRARY) $(OBJS1) $(OBJS2) $(OBJS3)
INCLS = -I../../../../src/lib/psmile/include \
        -I../../../../src/lib/mpp_io/include
INCLSC = -I../../../../src/lib/psmile/include \
        -I../../../../src/lib/mpp_io/include
.SUFFIXES:
.SUFFIXES: .o .f .F .f90 .F90 .c
%.o: %.F90
    $(F90) $(MPPFLAGS) $(F90FLAGS) $(INCLS) -c $<
%.o: %.f90
    $(f90) $(MPPFLAGS) $(f90FLAGS) $(INCLS) -c $<
%.o: %.F
    $(F) $(MPPFLAGS) $(FFLAGS) $(INCLS) -c $<
%.o: %.f
    $(f) $(MPPFLAGS) $(fFLAGS) $(INCLS) -c $<
%.o: %.c
    $(CC) $(CCFLAGS) $(INCLSC) -c $<
```

Appendix D

Library compile script generation tool

Script for the creation of the library compile script

```
#!/bin/ksh
#####
# File      : ~/util/compile/frames/Create_COMP_libs.frm
#
# Script to generate the library compile script
#
# Stephanie Legutke, MPI-HH, M&D          Dec 14, 2004
#
# This script is used to generate a compile script for the PRISM libraries.
# The script is specifically created for the target platform.
# Depending on the node name it selects site and OS specifics.
# The node name is either detected with 'uname -n' or, if the script is not
# created on the target machine, it must be specified by the user.
# The generated compile script is moved to ~/src/mod/'model name'
# after successful generation.
# The script uses m4 for preprocessing. It has been tested with
# GNU m4 version 1.4. Other, older or non-GNU versions of m4 may fail!
#
# This script is called interactively only.
#
# 5 positional parameters are possible:
#
# Usage : COMP_libs [- [- [node [MPI1]]]]
#
#     $1="/"-"-":          directing std output to a file / the screen
#     $2="/"-"-"/"+":      directing err output to a file / the screen / stdout
#     $3="/"-"node":        node name for header files with OS/site specs
#     $4="/"-"MPI1":        message passing
#
# If not called with all parameters, the defaults are:
#     $1 the gmake of the generated script directs stdout to a file
#     $2 the gmake of the generated script directs errout to a file
#     $3 node name of the machine where this scripts is run;
#         (if it is not the node name appearing in the header file
#         names with the site/OS specifications, it must be given)
#     $4 MPI2
#####
#
echo
echo '-----'
echo ' '
echo ' This script may fail if you do not use the right version of m4!'
echo ' The version you use is : ' `m4 --version`
echo ' Make sure this is GNU m4 younger than version 1.4!'
echo ' '
echo '-----'
set +x
if [ "$1" = "--" ] || [ "$1" = "help" ] || [ "$1" = "--help" ] ; then
echo
echo '----- Usage -----'
echo ' $1="/"-"-":          directing std output to a file / the screen'
echo ' $2="/"-"-"/"+":      directing err output to a file / the screen / stdout'
echo ' $3="/"-"node":        node name for header files with OS/site specs'
echo ' $4="/"-"MPI1":        message passing'
echo '-----'
echo ' '
echo
exit
fi
echo
# Redirecting standard output and error output
#
outout='stdout=$SCRIPTDIR/COMP_libs.log'
if [ "$1" != "" ]; then
if [ "$1" = "-" ]; then
```

```

        outout='stdout='tty''
        echo '* (g)make standard output will directed to screen.'
    else
        echo 'This values is not allowed for parameter 1 (std out)'
        exit
    fi
else
    echo '* (g)make standard output will be directed to a file.'
fi
errout='stderr=$SCRIPTDIR/COMP_libs.err'
if [ "$2" != "" ]; then
    if [ "$2" = "-" ]; then
        echo '* (g)make standard error will directed to screen.'
        errout='stderr='tty''
    elif [ "$2" = "+" ]; then
        echo '* (g)make standard error will directed to stdout.'
        errout='stderr="$stdout"'
    else
        echo 'This values is not allowed for parameter 2 (std err)'
        exit
    fi
else
    echo '* (g)make standard error will be directed to a file.'
fi
#
# HOST, OS, node name for header file directory selection
#
node='uname -n'
build='uname -s'
if [ "$3" != "" ]; then
    if [ "$3" != $node ]; then
        echo 'Warning: the node name specified is not the node name of this machine'
        echo 'Ensure that the site/OS specific header files exist.'
        node=$3
    fi
fi
echo ' '
echo '* Creating library compile script'
echo '      on/for a '$build' platform with node name '$node'.'
echo ' '
node1_4='echo ${node} | cut -c1-4'
if [ ${node1_4} = xbar ]; then node=${node1_4}; fi
if [ ${node1_4} = hpca ]; then node=${node1_4}; fi
node1_2='echo ${node} | cut -c1-2'
if [ ${node1_2} = ds ]; then node=${node1_2}; fi
node1_6='echo ${node} | cut -c1-6'
if [ ${node1_6} = total1 ]; then node=${node1_6}; fi
echo ' '
echo '* Abbreviated node name is '$node'.'
echo ' '
mespas=irgendwas
messpass=MPI2
if [ "$4" != "" ]; then
    if [ "$4" = MPI1 ] || [ "$4" = NONE ] ; then
        messpass=$4
        mespas=MPI2
    else
        echo 'This may not be specified:'$4
        exit 1
    fi
else
    echo '* The message passing will be '$4'.'
fi
echo '* The message passing will be MPI2.'
fi
#
# Change to directory of script
#
scriptdir='dirname $0'
cd $scriptdir
# /u/fj/mfuj/mfuj004/bin/m4" ; xbar
cat > infile.m4 <<EOF
#!/bin/ksh
changequote([,])dnl
changeocom
dnl
dnl Embedding as batch (not needed on all nodes)
dnl
define(comp_Model,comp_libs)dnl
sinclude(include/Qsub_start_${node}.h)dnl
undefine([comp_Model])dnl
dnl
dnl Comments, usage, history
dnl
include(include/Comments_libs.h)dnl
include(include/Comments_libs_frm.h)dnl
dnl
dnl GUI input for all compile scripts
dnl
define(${mespas},${messpass})dnl
include(include/Guispecif_all.h)dnl
undefine([${mespas}])dnl
dnl
dnl GUI input depending on model
dnl
include(include/Guispecif_libs.h)dnl

```

```

dnl
dnl  Command parameter input (scripting only)
dnl
include(include/Command_par_libs_frm.h)dnl
include(include/Input_check_all_frm.h)dnl
include(include/Input_check_libs_frm.h)dnl
dnl
dnl  Parameter renaming and printing
dnl
include(include/Print_par_all.h)dnl
include(include/Print_par_libs.h)dnl
scriptdir='\dirname \$0\'
cd \$scriptdir
export SCRIPTDIR='\pwd\'
node='\uname -n\'
dnl
dnl  HOST, OS, node name
dnl
include(include/Prolog_all.h)dnl
include(include/Prolog_libs.h)dnl
export TARFILE=libs_\`date +%y%m%d\`.tar
dnl
dnl  Site and OS specifics
dnl
include(include_${node}/Sitespecific_${node}.h)dnl
include(include_${node}/Compile_mode_libs_${node}.h)dnl
include(include_${node}/OSspecific_${node}.h)dnl
dnl
dnl  Non Site and OS dependent cpp flags
dnl
include(include/Cppflags_libs.h)dnl
include(include/Cppflags_edit.h)dnl
dnl
dnl  Build the build
dnl
include(include/Build_dirs_libs.h)dnl
dnl
dnl  Create top level Makefile
dnl
include(include/Top_makefile_all.h)dnl
define(NodeName, $node)dnl
include(include/Top_makefile_libs.h)dnl
undefine(NodeName)dnl
define(outdev, ${outout})dnl
outdev
undefine([ {outdev} ])dnl
define(errdev, ${errout})dnl
errdev
undefine([ {errdev} ])dnl
if [ -f \$BLDROOT/lib.status ]; then rm \$BLDROOT/lib.status; fi
dnl
dnl  Make libraries
dnl
include(include/Make_libs.h)dnl
dnl
dnl  Print update status : scripting only
dnl
include(include/Status_libs_frm.h)dnl
dnl
dnl  Embedding as batch (not needed on all nodes)
dnl
sinclude(include/Qsub_end_${node}.h)dnl
exit
m4exit
EOF
#
# Run preprocessor: note the blank behind -D / -U
#
m4 infile.m4 > COMP_libs
status=$?
if [ $status -eq 0 ]; then
mv COMP_libs ../../COMP_libs.${node}
chmod 755 ../../COMP_libs.${node}
olddir=`pwd`
cd ../../
moddir=`pwd`
echo "* The compile script name is "$moddir/COMP_libs.${node}
echo ' '
else
echo "An error occurred! Status="$status
echo ' '
fi
cd $olddir
rm infile.m4
exit

```

Appendix E

The library compile script

**Compile script used for the compilation of all libraries
(with MPI-2 on a SX NEC machine with node name ds)**

```
#!/bin/ksh
#####
#
#   C O M P I L E - script for maintaining PRISM libraries.
#
#   File       : ~/util/COMP_libs
#
#   This script is used to compile or clean any PRISM library.
#   This include libraries used exclusively in any of the models
#   or those used by the coupler only, or those shared by the models.
#
#   When needed, different versions of the libraries are created.
#   At the moment these are MPI1/MPI2 libraries.
#
#   This script is called directly by the administrator or SMS
#   or by the model's compile scripts (on scripting level).
#   When called by the model's compile scripts it checks and updates
#   the libraries needed to compile the models and only those.
#
#####
#
#   4 positional parameters are possible:
#   $1=make_target      (optional)
#   $2=list_of_libraries (optional)
#   $3=message_passing  (optional)
#   $4=calling_model    (optional)
#
#   Usage : COMP_libs      ["make_target" ["list_of_libraries" \
#                           ["message_passing" \
#                           ["calling_model"]]]]
#
#   If called with no parameters, the defaults given below
#   are used. If parameter n is specified; the parameter 1,...n-1
#   must be specified as well.
#   Note: parameter 4 is not used on the scripting level.
#
#####
set -e
##### Start of user specifications (defaults) #####
#
# message_passing          : NONE / MPI 2/1
message_passing=MPI2
# coupler                  : oasis3
# use_key_noIO             : whether mpp_io will be used for I/O
#
coupler=oasis3
use_key_noIO=no
# list_of_libraries       : any list of PRISM libraries possible
# compile mode            : default/opt/..
compile_mode=default
use_NetCDF=yes
list_of_libraries=" anaisg anaism clim fscint mpp_io psmile scrip"
# make_target              : clean/all/tar/lib
make_target=all
##### Command line parameter #####
#
# Help function
#
if [ "$1" = "--" ] || [ "$1" = "help" ] || [ "$1" = "--help" ] ; then
  echo
  echo '----- Usage -----'
  echo ' $1=make_target      (all/tar/clean; optional)'
  echo ' $2=list_of_libraries (anaisg/anaism/clim/etc.; optional)'

```

```

echo ' $3=message_passing (NONE/MPI1/MPI2; optional)'
echo ' '
echo ' Default active values are:'
echo ' make_target : '${make_target}'
echo ' list_of_libraries : '${list_of_libraries}'
echo ' message_passing : '${message_passing}'
echo '-----'
exit 1
fi
# Overwrite with command line parameter if specified
[ "$1" = "" ] || make_target=$1
[ "$3" = "" ] || message_passing=$3
#
# if no message passing is specified, only compile the interpolation libraries
#
if [ "${message_passing}" = "NONE" ]; then
list_of_libraries=" anaimg anaism fscint scrip"
fi
[ "$2" = "" ] || list_of_libraries=$2
set -u
##### Input checks : all #####
#
#
if [ "${make_target}" != "clean" ] && \
[ "${make_target}" != "all" ] && \
[ "${make_target}" != "lib" ] && \
[ "${make_target}" != "tar" ]; then
echo " Invalid target : "${make_target}"
echo " The task is stopped!"
exit 1
fi
if [ "${message_passing}" != "MPI2" ] && \
[ "${message_passing}" != "MPI1" ] && \
[ "${message_passing}" != "NONE" ]; then
echo " Invalid option for message passing : "${message_passing}"
echo " The task is stopped!"
exit 1
fi
##### Input checks : libraries only #####
#
for library in ${list_of_libraries}; do
if [ "${library}" != "anaimg" ] && \
[ "${library}" != "anaism" ] && \
[ "${library}" != "blas" ] && \
[ "${library}" != "clim" ] && \
[ "${library}" != "fscint" ] && \
[ "${library}" != "ioips1" ] && \
[ "${library}" != "lapack" ] && \
[ "${library}" != "linpack" ] && \
[ "${library}" != "mpp_io" ] && \
[ "${library}" != "netcdf90" ] && \
[ "${library}" != "psmile" ] && \
[ "${library}" != "scrip" ] && \
[ "${library}" != "libV4" ] && \
[ "${library}" != "svipc" ] && \
[ "${library}" != "support" ]; then
echo " Invalid library : "${library}"
echo " The task is stopped!"
exit 1
fi
done
##### Set and print input parameter: all #####
#
export COUPLER=${coupler}
export MAKETARGET=${make_target}
echo " "
echo "Make target : "${MAKETARGET}
export CHAN=${message_passing}
##### Set and print input parameters: libs #####
#
export srclibs=${list_of_libraries}
echo " "
if [ "${MAKETARGET}" = "all" ]; then
echo "Libraries which will be updated : "
echo $srclibs
elif [ "${MAKETARGET}" = "clean" ]; then
echo "Libraries which will be cleaned : "
echo $srclibs
elif [ "${MAKETARGET}" = "tar" ]; then
echo "A tar-file of libraries will be created with:"
echo $srclibs
fi
#####
#
# Libraries for all models
#
TARFILE_DIRS=""
for srclib in $srclibs; do
TARFILE_DIRS=${TARFILE_DIRS}" src/lib/${srclib}"
done
export TARFILE_DIRS
scriptdir=`dirname $0`
cd $scriptdir
export SCRIPTDIR=`pwd`
node=`uname -n`

```

```

##### Prolog #####
#
# 'Prolog' : Node name and operating system etc.
#
export NODE=$node
echo ' '
echo 'This script runs on node ' $NODE'.'
cd $SCRIPTDIR
cd ..
export SRCROOT=`pwd`
# echo 'PRISM top source root directory is '$SRCROOT
export TARFILE=libs_`date +%y%m%d`.tar
##### Site and OS dependent specifications #####
#
#           ds      : part of name of header file
#           NODE    : where the model is      compiled
export NODE=`echo ${NODE} | cut -c1-2`
if [ ${NODE} != ds ]; then
    echo ' The OS/Site dependent specifications are not for this machine.'
    echo ' Node name of this machine:' ${NODE}
    echo ' Script setup is for use on a ds node (cross compiling for SX)'
if [ "$MAKETARGET" != "tar" ]; then
    echo ' The task is stopped!'
    exit 1
fi
fi
if [ "$MAKETARGET" = "lib" ]; then
    echo ' '
    echo 'No executable will be created. '
fi
export ARCH=SX
export BLDROOT=$SRCROOT/$ARCH
set +u
if [[ -z "$LD_LIBRARY_PATH0" ]]; then
    LD_LIBRARY_PATH0=""
fi
set -u
. /SX/opt/etc/initsx.sh
export MPIROOT=${SX_BASE_MPI}
export MPI_LIB="-L${SX_BASE_MPI}/lib0 -lmpi"
export MPI_LIB=""
export MPI_INCLUDE=${SX_BASE_MPI}/include
export NETCDFROOT=/pool/SX-6/netcdf/netcdf-3.5.0
export NETCDF_LIB="-L${NETCDFROOT}/lib -lnetcdf_c059_f285"
export NETCDF_INCLUDE=${NETCDFROOT}/include
export LIB1="-L/SX/opt/MathKeisan/inst/lib -llapack -lblas -llinpack"
export LIB2="-L/pool/SX-6/NAG/fnne504db -lnag "
export LIB3=""
export SYS_INCLUDE=${SX_BASE_CPLUS}/include/C++
export SYS_INCLUDE=./
if [ "$MAKETARGET" = "all" ] || [ "$MAKETARGET" = "lib" ]; then
    echo ' '
    echo 'Compiler revision : '
    echo ` ${SX_BASE_F90}/bin/sxf90 -V `
fi
export F90com=${SX_BASE_MPI}/bin/sxmpif90
export f90com=${F90com}
export Fcom=${F90com}
export fcom=${F90com}
export cc=${SX_BASE_MPI}/bin/sxmpic++
export ar=${SX_BASE_CROSS}/bin/sxar"
export as="sxas"
export cp=cp
PATH=$SRCROOT/util:$PATH
alias make="gmake"
# Compile modes (specified by user): libraries on a ds* node (cross/cumulus)
#
#       Note : the libs must be searched cyclically now for oasis3!
export OPENMP=no
if [ $compile_mode = default ]; then
    optc=""
    optl="-Wl,-h lib_cyclic"
    optf=" -Wf,-pvctl noassume loopcnt=5000000 "
elif [ $compile_mode = debug ]; then
    echo "not yet possible"
    exit 1
elif [ $compile_mode = profile ]; then
    echo "not yet possible"
    exit 1
elif [ $compile_mode = opt ]; then
    optc="-Chopt"
    optf="-Chopt"
    optl="-Wl,-Z 1000000 -pi line=1000"
elif [ $compile_mode = ad_hoc ]; then
    echo "with this option the options are set through GUI input"
    exit 1
else
    echo "Invalid compile mode"
    exit 1
fi
#
# All models should use cpp flag __SX for activation of platform dependent
# source code on SUPER-SX machines !
#

```

```

# `uname -n` : ds (cross compiling only); cs17
# `uname -s` : SUPER-UX
# `uname -m` : SX-6
#
# With old compiler revision which do not allow for the
# option  optdbl="-Wf,-A idbl4" use:
#         optdbl="-Wf,-A db14" AND -D__SXdbl4
#
export CPPFLAG="-D__SX -D__SX__ "
export CPPFLAGF90start=""
export CPPFLAGSEP=" "
optdbl="-Wf,-A idbl4"
lstflags="-Wf,-L fntlist transform"
if [ "$OPENMP" = "yes" ]; then
    Popt=openmp
    CPPFLAG=${CPPFLAG} -D__openmp -DUSE_OMP"
else
    Popt="stack"
fi
export F90FLAG="-EP -P$Popt ${optf} ${optl} ${optdbl} ${lstflags} "
export FFLAG=${F90FLAG}
export f90FLAG=${F90FLAG}
export fFLAG=${F90FLAG}
export LDFLAG="-P$Popt ${optf} ${optdbl}          ${optl} "
export CCFLAG=${optc}
export AFLAG="-h float0 -m"
export AFLAG="-float0 -e b" #pw
export AFLAG="-h sx6 -m"
export ARFLAG=" ruv"
export MPPFLAG=" "
export I4mods="I"
#####
# Libraries CPP flags not depending on site or OS
#
CPPFLAG=${CPPFLAG} -Duse_comm_${CHAN} -Duse_libMPI"
if [ ${use_key_noIO} = yes ]; then
    CPPFLAG=${CPPFLAG} -Dkey_noIO"
fi
if [ ${use_NetCDF} = yes ]; then
    CPPFLAG=${CPPFLAG} -Duse_netCDF"
fi
CPPFLAG=${CPPFLAG} -DNC_DOUBLE"
export CPPFLAGCC=${CPPFLAG}
CPPFLAGF90=""
for flag in $CPPFLAG ; do
    CPPFLAGF90=${CPPFLAGF90}${CPPFLAGSEP}${flag}
done
if [ "$CPPFLAGF90" != "" ]; then
    CPPFLAGF90=${CPPFLAGF90start}${CPPFLAGF90}
fi
export CPPFLAGF90
if [ "$MAKETARGET" = "all" ] || [ "$MAKETARGET" = "lib" ]; then
    echo '
    echo "CPP flags used with the cc command: "
    echo $CPPFLAGCC
    echo '
    echo "CPP flags used with the f90 command: "
    echo $CPPFLAGF90
fi
##### Directories #####
# Container 'Build the build' : make build directories if needed
#
libs=${BLDROOT}/lib
[ -d $libs ] || mkdir -p $libs
lib=${BLDROOT}/build/lib
BLDLIBS=" "
bldlib=""
for library in $srclibs; do
    if [ $library = psmile ] || \
       [ $library = clim ] ; then
        bldlib=${library}.${CHAN}
    else
        bldlib=${library}
    fi
    if [ ! -d ${lib}/${bldlib} ]; then
        echo 'Making build directory '${lib}/${bldlib}' ...'
        mkdir -p ${lib}/${bldlib}
    fi
    ${cp} -p ${SRCROOT}/src/lib/$library/src/Makefile ${lib}/${bldlib}
    BLDLIBS=${BLDLIBS} " ${bldlib}
done
export BLDLIBS
cd ${lib}/${bldlib}
#####
# Create the top level Makefile
#
cat > Top_Makefile_$$ <<'EOF1'
export
SHELL = /bin/ksh
builddir = ./
prefix = ../../../../

```

```

includedir = ${prefix}/include
configdir = ${prefix}/config
exec_prefix = ../../..
top_builddir = $(exec_prefix)/build
bindir = $(exec_prefix)/bin
libdir = $(exec_prefix)/lib
INCLUDES = -I$(NETCDF_INCLUDE) -I$(MPI_INCLUDE) -I$(SYS_INCLUDE)
F90       = ${F90com}
f90       = ${f90com}
F         = ${Fcom}
f         = ${fcom}
CC        = ${cc}
CPP       = ${cc} -E
AR        = ${ar}
AS        = ${as}
AFLAGS    = $(AFLAG)
ARFLAGS   = $(ARFLAG)
CCFLAGS   = -I$(configdir) -I$(SYS_INCLUDE) $(CCFLAG) $(CPPFLAGCC)
F90FLAGS  = $(F90FLAG) $(INCLUDES) $(CPPFLAGF90)
FFLAGS    = $(FFLAG) $(INCLUDES) $(CPPFLAGF90)
f90FLAGS  = $(f90FLAG) $(INCLUDES) $(CPPFLAGF90)
fFLAGS    = $(fFLAG) $(INCLUDES) $(CPPFLAGF90)
COUPLE    = ${COUPLER}
MPPFLAGS  = $(MPPFLAG)
LIBDIRS   = ${BLDLIBS}
all:
@for DIR in $(LIBDIRS) ;\
do \
    back=`pwd` ; \
    echo ` ` ; \
    cd ../../lib/$$DIR ; \
    $(MAKE) -f Makefile $(MAKETARGET); status=$$? ; \
    echo $$DIR `:` $$status >> $(BLDROOT)/lib.status ; \
    if [ $$status != 0 ] ; then \
        echo "Exit status from make was $$status" ; \
    fi ; \
    cd $$back ; \
done
clean:
@for DIR in $(LIBDIRS) ;\
do \
    back=`pwd` ; \
    echo ` ` ; \
    cd ../../lib/$$DIR ; \
    $(MAKE) -f Makefile $(MAKETARGET); status=$$? ; \
    echo $$DIR `:` $$status >> $(BLDROOT)/lib.status ; \
    if [ $$status != 0 ] ; then \
        echo "Exit status from make was $$status" ; \
    fi ; \
    cd $$back ; \
done
tar:
cd $(prefix) ; \
tar -cvf ${TARFILE} \
${TARFILE_DIRS} \
util/compile
EOF1
# util/COMP_libs.ds \
stdout=$SCRIPTDIR/COMP_libs.log
stderr=$SCRIPTDIR/COMP_libs.err
if [ -f $BLDROOT/lib.status ] ; then rm $BLDROOT/lib.status; fi
#####
# Execute makefile : -p : listing of all default rules
#                  -d : debug
#                  -k : dont stop when error occurs
#
mv Top_Makefile_$$ ../Top_Makefile_$$
if [ -f ${stdout} ] ; then
    rm -f ${stdout}
fi
if [ -f ${stderr} ] ; then
    rm -f ${stderr}
fi
make -f ../Top_Makefile_$$ $MAKETARGET -k 1>${stdout} 2>${stderr}
rm ../Top_Makefile_$$
#
# Print compilation status
#
if [ $MAKETARGET != "tar" ] ; then
    echo ` `
    cat $BLDROOT/lib.status
    set +e
    grep -v `:` 0 $BLDROOT/lib.status
    if [ $? != 0 ] ; then
        echo 'Libraries ...'
        echo $srclibs
        if [ $MAKETARGET = all ] ; then echo '... are up-to-date!' ; fi
        if [ $MAKETARGET = clean ] ; then echo '... are cleaned!' ; fi
        echo ` `
    else
        if [ $MAKETARGET = all ] ; then echo 'Updating libraries '$srclibs' failed !' ; fi
    fi
fi

```

```
    if [ $MAKETARGET = clean ]; then echo 'Cleaning libraries '$srclibs' failed !'; fi
    echo 'Compilation is stopped!'
    exit 1
fi
set -e
fi
echo ' '
#
#####
exit
```