Consortium

for

Small-Scale Modelling

COSMO Standards for Source Code Development

Version 1.0

Ulrich Schättler

August 2011

Deutscher Wetterdienst

MeteoSwiss

Ufficio Generale Spazio Aereo e Meteorologia

ΕΘΝΙΚΗ ΜΕΤΕΩΡΟΛΟΓΙΚΗ ΥΠΗΡΕΣΙΑ

Instytut Meteorologii i Gospodarki Wodnej

Administratia Nationala de Meteorologie

Росгидромет

Agenzia Regionale per la Protezione Ambientale del Piemonte

Agenzia Regionale per la Protezione Ambientale dell' Emilia-Romagna: Servizio Idro Meteo Clima

Centro Italiano Ricerche Aerospaziali

Amt für GeoInformationswesen der Bundeswehr

www.cosmo-model.org

Editor: Ulrich Schättler

Revision	Modifications	Date	Author
0.1	Initial Draft	August 2010	U. Schättler DWD
0.2	Revision of chapters; Inserted *Procedure for changing the Reference Version* into Chapter 6.	January 2011	U. Schättler DWD
0.3	Revision of (mainly) Chapter 6 Incorporated suggestions from all colleagues Adopted phrases from CLM, who take over these standards.	April 2011	U. Schättler DWD
0.4	Compatibility with other COSMO Software Added Appendix for `fieldextra` Incorporated some more changes regarding CLM Community	July 2011	J.M. Bettems MeteoSwiss U. Schättler DWD
0.5	Revision of Section 2 Amendments to Section 6	August 2011	U. Schättler DWD
1.0	Revisions by the SPM Marco Arpagaus	August 2011	U. Schättler DWD

Some Abbreviations and Explanations

STC	Steering Committee The STC is the main governing and steering body of COSMO. It is responsible for and approves the changes to the official versions of the COSMO Software.
SMC	Scientific Management Committee The SMC defines the changes to the COSMO Software and is responsible for their implementation.
TAG	Technical Advisory Group The TAG defines the technical rules for implementing COSMO Software.
SCA	Source Code Administrator Every COSMO Software has a main SCA, who is responsible for the maintenance.
VCS	Version Control System A system to manage changes to source code and documentation which allows to track modifications and retains the history of all previous versions of a software.

Contents

1 Introduction **1**

2 The Development Process - Some Mandatory Aspects **3**

3 Software Design **6**

4 Coding Rules **8**

4.1 Style and Typing Rules 8

4.2 Mandatory Features 11

4.3 Banned Features 11

5 Documentation **13**

5.1 Process Documentation 13

5.2 Product Documentation 14

5.3 Delivering New or Modified Code 15

6 Software Maintenance and Quality Control **16**

6.1 Version and Release Management 16

6.2 Rules for Implementation of Changes 18

6.3 Rationale for Changes 21

6.4 Release Planning 22

6.5 Standard Test Suite for the COSMO-Model 22

6.6 Meteorological Test Suite 23

6.7 Testing the official versions in the VCS 23

7 Implementation Issues **25**

7.1 Memory Management 25

7.2 Working Precision 26

7.3 Parallelization 26

7.4 Optimization 26

7.5 Vectorization 26

7.6 I/O 26

7.7 Error Handling / Debug Messages 26

A Standard Headers for the COSMO-Model and INT2LM 29

B Amendments for `fieldextra` 33

1 Introduction

This document specifies standards for source code development within the *Consortium for Small Scale Modeling*, COSMO. The CLM Community, which uses the COSMO-Model, also follows these standards. They are valid for all versions of the COSMO Software and will also be applied to software provided by other institutions or communities that is intended to be incorporated into COSMO Software. At the date of writing the COSMO Software is composed of

- the COSMO-Model,
- the interpolation program `INT2LM`,
- the postprocessing tool `fieldextra`,
- the verification package `VERSUS`.

The COSMO Standards for Source Code Development are based on the *European Standards for Writing and Documenting Exchangeable Fortran 90 Code* [1] and should complement and adapt these for the needs of COSMO where necessary. The *European Standards* have been defined in the 90ies to facilitate the exchange of code between meteorological centres. They provide a framework for the use of Fortran 90, in which some details have to be elaborated.

The exchange of code surely has not become common practice up to now, but following the *European Standards* facilitated the development and documentation of the COSMO-Model. This, in turn, lead to the dissemination of the model system to a wide community. The further development of the COSMO-Model within this bigger community now requires an even stronger emphasis on a clear and structured development and a detailed documentation. This also holds for the other components of the COSMO Software.

The aim of this document is to provide some guidelines for several aspects of the development and programming process and should serve the following goals:

- encourage well structured programming
- increase readability and useability of source code
- standardize the look and usage of source code units
- create machine independent portable source code
- simplify maintenance tasks
- facilitate quality management and control

The main chapters of the COSMO Standards are:

- *Mandatory Aspects*
  Throughout all sections of this paper, some mandatory aspects of source code development are specified. This section just summarizes the most important ones at a glance and can be used as a step-by-step instruction.

- *Software Design*
  Software design is the process of problem solving and planning a software solution. It thus includes the construction of the underlying physical model as well as the definition of the major components of the software needed to provide a robust and efficient solution.

- *Coding Rules*
  The coding rules specify a common programming style that has to be adhered to by all COSMO Software. This enhances the readability and exchangeability of source code.

- *Documentation*
  The term *Documentation* refers to both kinds of documentation, i.e. the process documentation, which can be defined separately for every project, and the product documentation. The product documentation is split into the internal code documentation and the external documentation outside of the code.

- *Software Maintenance and Quality Control*
  An essential part of the software development process is the maintenance of the software and the quality control of each version. Also, clear rules have to be defined how to go from one version to another in operational production.

- *Implementation Issues*
  In addition to the Coding Rules, we will give some recommendations for the practical implementations. Issues like memory management, parallelization, optimization and vectorization will be discussed briefly.

The preferred language for model development still is Standard Fortran. But as new computer architectures evolve, it might be necessary to use other languages or language extensions. Especially for non-numeric applications (like VERSUS) Fortran surely is not the first choice for the computer language to use. But no matter which language is used, a *portable*, i.e. a *standard* source code is absolutely necessary. Therefore we intend to transfer these standards *logically* also to non-Fortran applications.

We are aware of the fact that the existing COSMO Software does not fulfill all of these standards, and sometimes not even the most necessary ones. But the development team is working on that and tries to make the COSMO-Model better with every new version. Not only regarding the meteorology, but also regarding the software engineering aspects.

Some rules stated in this document do not apply to all of the COSMO Software. This will be explicitly stated when this is the case. If additional rules or clarifications are necessary, they are stated in appendices to that document.

Note that additional and / or slightly different and adjusted rules might be valid for the source code development at partner institutions and communities for source code not to be incorporated into COSMO Software.

This document will be enacted during summer 2011, but will be revised in the future based on the experiences made. The users are invited to give their feedback.

2 The Development Process - Some Mandatory Aspects

This section sketches the process of software development, i.e. the steps that modifications or new contributions to COSMO Software have to pass, before they are part of official COSMO Software. It summarizes the general rules that have to be fulfilled by the developers and can be taken as step-by-step instructions, how COSMO Software can be developed. Links to more detailed explanations in the following chapters are given.

For a better understanding some terms are defined first.

- **The Software:**
  The code under consideration.
- **The Developer:**
  Everybody contributing to the Software by implementing changes and / or new components.
- **The Management:**
  The group that has to decide on changes to the Software. For COSMO this is the Scientific Management Committee (SMC) in collaboration with the Steering Committee (STC). For the CLM Community it is the CLM Coordination group (CLM_CO).
- **Private versions and official versions:**
  The Software is maintained using a version control system (VCS). Every version available by such a VCS is an official version, while all other versions, that exist only in a private framework, are private versions.
- **The Source Code Administrator:**
  Every COSMO Software has a main Source Code Administrator (SCA), who is responsible for the maintenance.

In the following, the steps are specified that changes to the Software have to pass. At the same time, it shows the management decisions that have to be taken when adopting developments to the official COSMO Software.

1. **The idea:**
   In the beginning there has to be an idea about what can be developed for the Software. The Developer implements all necessary changes into a private version based on the latest official version of the Software and performs appropriate tests. He / she presents the results as a report and / or a presentation to the Management and proves the usefulness of the changes.

   Note:
   In this state it is not necessary to fulfill the rules of the COSMO-Standard, but it will be very helpful for the further steps.

2. **The first decision:**
   The Management decides whether the idea and the corresponding change to the Software is in-line with the scientific and / or the technical planning. If it is of interest and usefulness has been proved, it is included into the *Planning for Future Developments*. (see Chap. 6.4, Release Planning, for details).
   The Developer is asked to design and implement the changes according to the COSMO-Standards. Depending on the complexity of the changes the Management asks the Developer to perform appropriate tests.

   Note:
   Ideas can be initiated (among others) by individuals, the Working Groups, and the Management itself.

3. **The development:**
   The Developer now has to prepare the changes and do the tests according to the COSMO Standards. This is still done with a private version. The following issues have to be met:

   - **The source code provided is properly designed:**
     Some guidelines for software design (especially for the COSMO-Model) are outlined in Chap. 3, Software Design. The Developer should design the changes and the assembly into the official Software in collaboration with the SCA and the code responsible persons of the corresponding software components.

   - **The source code provided conforms to the coding rules:**
     The decision, whether the coding rules are fulfilled, is taken by the SCA of the corresponding software. In case of conflict, the case is taken to the *Technical Advisory Group* (TAG).
     For CLM versions, the CLM Community has its own TAG, which is responsible in this case.
     (See Chap. 4, Coding Rules, for details)
   - **All source code modifications are documented:**
     This means that for all modifications (extensions, changes, bug fixes) the product documentation (scientific, user guide) has been written or updated and a contribution to the process documentation (short description of the changes for the version log-file) is available.
     (See Chap. 5, Documentation, for details).
   - **All changes have been tested. The results are published appropriately:**
     To ensure continuing high quality of the COSMO Software, tests have to be performed for all new or modified Software. Depending on the scope of the changes, more or less extensive tests are required. The results have to be published to inform the community (e.g. as a COSMO Technical Report, a contribution to the COSMO-Newsletter, a presentation at the General Meeting or the COSMO User Seminar).
     (See Chap. 6, Software Maintenance and Quality Control, for details)
   - **A *Code Responsible Person* is available in the future:**
     In order to assure a good quality code and support in the future, there is the

need that all components of the code do have a responsible person, who can be contacted in case of questions or problems.
(See Chap. 6, Software Maintenance and Quality Control, for details)

4. **The second decision:**
The Management checks the results of the tests and whether the above conditions are fulfilled. If all issues are satisfied, the change is included into the *Release Planning*, which defines the intended version number and date for an official release. The code is given to the SCA for implementation into an official version.

5. **The official implementation:**
The SCA implements the changes into an official version. As there might be several contributions, which should not be implemented all in one, there might be several consecutive versions, which are not released officially (called Development Versions, see 6.1, Version and Release Management, for more details). The SCA provides the officially implemented code to the Developer again for cross-checking.

3 Software Design

Before the software can be designed, the physical model has to be constructed. For non-physical (or non-numerical) projects an abstract specification has to be developed that specifies the tasks of the software and the services it should provide. Also the constraints under which it has to operate have to be defined. A *Top-Down* design, starting by the main task and splitting this into smaller units (modular design), is most promising for an efficient software development. Depending on the complexity of the model, the system can be split into subsystems, which themselves consist of components. The components then have to be mapped to programming language components such as procedures, functions or modules[1].

Fig. 1 gives a sketch of the top-down design for the software of the COSMO-Model. This rather coarse but generic breakdown could also be used for other software projects.

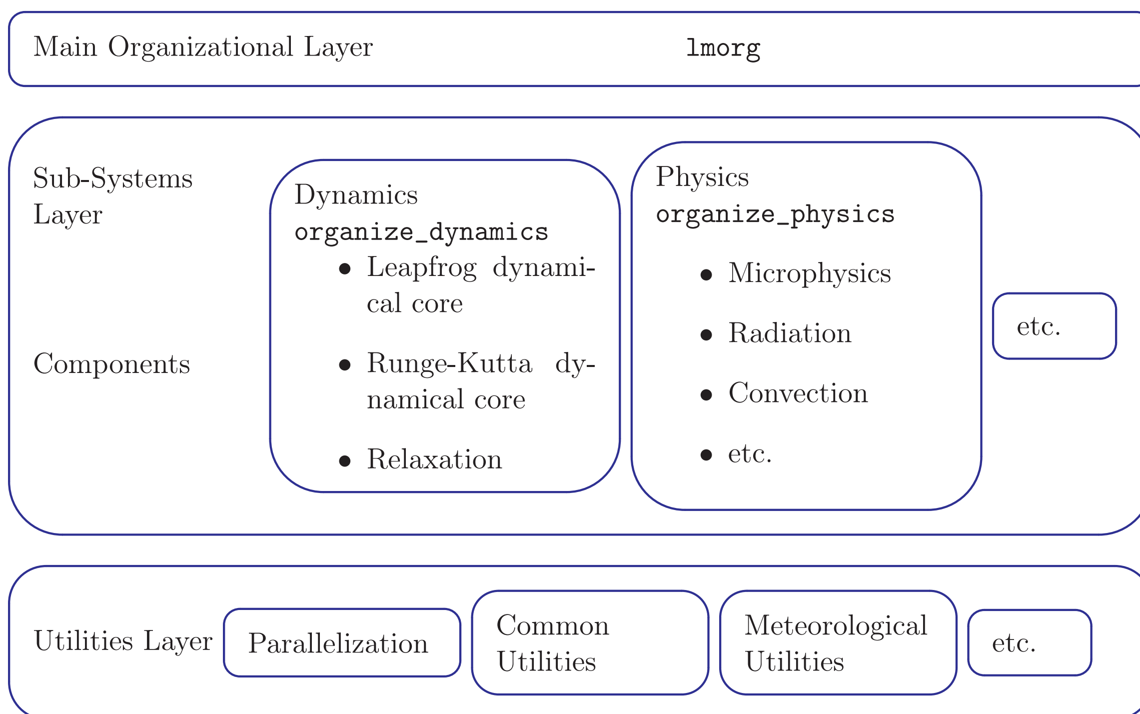

Figure 1: Schematic view of a Software Design

In the following we will give some common rules for the design and implementation of individual software components, especially for the use in meteorological models. Some of them are derived from the *European Standards* [1], which in turn are based on the plug-compatibility rules of Kalnay [2]. She defines a *package* as a set of Fortran subroutines to parameterize atmospheric subgrid-scale physical processes. To exchange packages between different centres it is necessary to use a standard programming style that facilitates the interchange.

[1] In the *European Standards* the term *package* is used, but not cleary defined. As they refer to the *plug compatibility* rules of Kalnay et al. [2], it is likely that e.g. the physics packages are meant. More commonly we will speak of software components (or just components).

"The goal of these rules is to make the "physics" routines easily transferable between models with only a few hours of work, and at the same time minimize the degradation in model efficiency."

Common rules for the design and implementation

- A package / component shall provide different set-up and running procedures (if necessary), each with a single entry point. All initialization of static data must be done in the set-up procedure and these data must not be altered by the running procedures.
- A package / component shall refer only to
    - its own modules and subprograms,
    - to programmer-defined utility procedures such as error-handling, parallelization or basic algorithms,
    - to external libraries,
    - to the intrinsic routines included in the Fortran standard.
- The package / component shall not terminate program execution. If any error occurs within the package it should gracefully exit, externally reporting the error via an integer variable in its argument list (with the possibility to indicate fatal errors). Packages should also write a diagnostic message describing the problem, using Fortran I/O, to an 'error stream' unit number selectable via the package's set up routine.
- The package / component should be written as general as possible (e.g. independent of resolution, time step, etc.) The resolution, for example, must be adjustable via the set up routine for the package.
- The package / component should be written in such a form that it can easily be extended in the future, if e.g. the requirements change.
- The routines of a package / component shall be grouped together in bigger program units of reasonable size. In principle it would be good to have them all in one file (or compilation unit), but this might not be possible due to computer resources (available memory during compilation) or also human resources (one file of several thousand lines of code will be hard to manage by a programmer).
- The interface(s) with other packages / components must be clearly defined and documented. The interface specification must be unambiguous as it allows the package / component to be used without deeper knowledge of the internal operations.
- The data structures used in the system implementation must be defined and documented in detail. This also includes the memory management of the software.
- The algorithms used to provide the services have to be designed and documented.

System Installation

Once the software has been designed and implemented, some means should be provided for the installation of the system. This also has to be documented clearly.

4 Coding Rules

The general ethos is to write portable code that is easily readable and maintainable. Code should be written in as general a way as possible to allow for modifications. In practice this means that coding will take a little longer. This extra effort is well spent, however, as maintenance costs will be reduced over the lifetime of the software.

The SCA of the corresponding code will supervise the programmers and check all developments. If changes are not in line with the coding rules (or more general: with the COSMO-Standards), the code can be rejected and / or be given back to the developers for improvement. In case of conflict, a *Technical Advisory Group* (TAG) will decide on the issue. Since the rules specified in the following do not cover all aspects, the SCA and / or the responsible TAG (either COSMO or CLM) can be contacted in case of further questions.

The rules presented in this chapter are mainly applicable for Standard Fortran, but some of them can surely be transferred to other programming languages. We give some style and typing rules and for Fortran we distinguish between mandatory features and banned features from former Fortran standards (Fortran77 and older).

The Fortran Standard is evolving with the years but compilers do not always comply with the full standard. By the time of writing this document, the NEC Compiler for example does not support a significant part of the Fortran 2003 Standard. Therefore we only require compliance with the Fortran 95 Standard. Please contact the TAG before using elements from a newer standard.

These rules are primarily intended for the COSMO-Model and the INT2LM. Other COSMO Software might define additional or modified rules and conventions for development. See Appendix B for amendments regarding `fieldextra`.

4.1 Style and Typing Rules

The general objective behind a style guide is to write portable code that is easily readable and can be maintained by different people. Many rules follow common sense and should be obvious. Although it is always tempting for scientists to restrict coding efforts to their specific application, we strongly urge you to try and write the code in a more general form. This will make it much easier to make modifications or couple your routine to others.

The following coding rules can be split into

1. *Rules* that should be applied and will be checked by the source code administrators before implementing changes to the official version. Some of these rules are strict (which is indicated below) and their application is mandatory!
2. *Conventions* that are not checked but are highly recommended to follow.

1. Rules

- (strict) Only use standard language elements to ensure a portable code.
- (strict) Use free format syntax.
- Text style
  - (strict) Fortran statements (keywords) must be written in upper case only.
  - (strict) Names of variables, parameters, subroutines, etc., must be written in lower case.
  - (strict) Never use a Fortran keyword as variable name!
  - (strict) Owing to the international user community, all naming of variables, modules, functions and subroutines as well as all comments must be written in English.
- Code indentation
  - (strict) To improve the readability of the code indent the lines within `DO`; `DO WHILE`; block `IF`; `CASE`; `INTERFACE`; etc. Indentation of 2 characters seems to be appropriate in most cases.
  - (strict) Where they occur on separate lines indent also internal comments like the code they refer to for reflecting the structure of the code.
  - (strict) Indent continuation lines to ensure that e.g. parts of a multi line equation or function / procedure call line up in a readable manner.
  - (strict) Do not use tab characters in your code. This will ensure that the code looks as intended when ported or printed.
  - (strict) Use blank space in the horizontal and vertical, to improve readability. In particular leave blank space between variables and operators, and try to line up related code into columns. Similarly, try to make equations recognizable and readable as equations.
- Variables and constants
  - (strict) `IMPLICIT NONE` must be used in all program units. This ensures that all variables must be explicitly declared, and hence documented. It also allows the compiler to detect typographical errors in variable names. One `IMPLICIT NONE` per program unit is enough.
  - Try to use meaningful variable names, or at least avoid use of cryptic names. A variable name in Fortran can have up to 31 characters (Fortran 90/95) and even up to 63 characters in Fortran 2003. Nevertheless we think that it is still useful to name the temperature as `t` and not as `temperature`. This notation is also closer to the formulas in the documentation.
  - Do not use the `DIMENSION` statement or attribute: declare the shape and size of arrays inside brackets after the variable name on the declaration statement. Use the saved space for documenting.
  - Always use the `::` notation, even if there are no attributes.
  - Declare the length of a character variable using the `(LEN = )` syntax.

  - There still is no default REAL datatype across all compilers. Most compilers implement the REAL datatype in 32 bit decision, but there can be exceptions. To improve portability between all compilers it is extremely useful to make use of KINDs to obtain the required numerical precision and range. A module should be written to define parameters corresponding to each required KIND. This module can then be USEd in every routine allowing all variables to be declared with an appropriate precision. See Sect. 7.2 (Working Precision) for more details

- Program Units and Interface Blocks

  - (strict) Always name program units and always use the END PROGRAM; END MODULE; END SUBROUTINE; END INTERFACE; etc. constructs, again specifying the name of the program unit.
  - Always use USE modulename, ONLY: to specify which of the variables, type definitions etc. defined in a module are to be made available to the USEing routine.
  - For variables in argument lists: Always use the attribute INTENT (IN|OUT|INOUT) to indicate usage of the variables.

- Miscellaneous:

  - Checking of return codes (e.g. from ALLOCATE, DEALLOCATE, OPEN, READ statements) helps in avoiding unexpected program behaviour and should be used whereever possible.
  - I/O: Separate the information to be output from the formatting information on how to output it on I/O statements. That is, do not put text inside the brackets of the I/O statement.
  - Use `>`, `>=`, `==`, `<`, `<=`, `/=` instead of .gt., .ge., .eq., .lt., .le., .ne. in logical comparisons. The new syntax, being closer to standard mathematical notation, should be clearer.
  - If array notation is used, show the array's shape in brackets to improve the readability, e.g.

```
array_1d(:) = array_1d_b(:) + array_1d_c(:),

array_2d(:,:) = scalar * another_array_2d(:,:).
```

2. Conventions

- All new code should be based on the templates given for different programming units (see A). Unused header lines should be removed.

- We recommend the following conventions for variable names:

  - INTEGER variables start with the letters i, j, k, m, n
  - Local INTEGER variables start with iz, jz, kz, mz, nz
    (loop indices i,j,k, etc. are an exception to this rule!)
  - LOGICAL variables start with the letter l

  - Local `LOGICAL` variables start with the letter `lz`
  - `CHARACTER` variables start with the letter `y`
  - Local `CHARACTER` variables start with the letter `yz`
  - All other variables can start with any other letter except the above ones.
  - Local `REAL` variables start with the letter `z`

- Do not put multiple statements on one line: this will reduce code readability.
- If necessary, introduce continuation lines by & at the end of the line to be continued. Do not use empty lines between continued lines, because some compilers cannot handle them and will abort. For viewing on terminals and for printers, about 90 characters per line should not be exceeded.
- CPP (the C Pre-Processor) keys are only used for
  - Differentiating codes in relation to computer architecture and compiler properties;
  - Controlling the access to external libraries.
  - Controlling the access to non-COSMO software packages as e.g. COSMO-ART.

4.2 Mandatory Features

The newer Fortran Standards offer some means to write code faster and much more portable than e.g. Fortran 77. Among them are

- Dynamic memory allocation
- The `KIND` concept to specify the working precision

The use of these features is mandatory. For a more detailed explanation see Chapter 7.

4.3 Banned Features

Some of the following sections detail features deprecated in or made redundant from Fortran 90 onwards. Others ban features whose use is deemed to be bad programming practice as they can degrade the maintainability of the code.

- `COMMON` blocks - use the declaration part of `MODULE`s instead.
- `EQUIVALENCE` - use `POINTER`s or derived data types instead to form data structures.
- Assigned and computed `GOTO`s - use the `CASE` construct instead.
- Arithmetic `IF` statements - use the block `IF`, `ELSE`, `ELSEIF` `ENDIF` construct instead.
- Labels
  - Labelled `DO` constructs - use (named) `ENDDO` instead.

  - I/O routine's `END` and `ERR` - use `IOSTAT` instead.
  - `FORMAT` statements: use `CHARACTER` parameters or explicit format specifiers inside the `READ` or `WRITE` statement instead.
  - Avoid any unused statement like a labeled `CONTINUE` not being jumped to.

- `GOTO`: Avoid `GOTO` by making use of `IF`, `CASE`, `DO WHILE`, `EXIT` or `CYCLE` statements.
- `PAUSE`
- `ENTRY` statements – a subprogram may only have one entry point.
- Fixed source form
- Avoid `FUNCTION`s with side effects, i.e. functions that alter variables in their argument list or in modules used, or functions which perform I/O operations. Although this is common practice in C, there are good reasons to avoid side effects. First, the code is easier to understand, if you can rely on the rule that functions do not change their arguments, second, some compilers generate more efficient code for *pure* functions, because they can store the arguments in different places (from Fortran 95 on there are the attributes `PURE` and `ELEMENTAL`).
- Avoid changing the shape of an array implicitly when passing it into a subroutine. Although actually forbidden in the Fortran 77 standard it was very common practice to pass n dimensional arrays into a subroutine where they would, say, be treated as a 1 dimensional array. This practice, though banned in Fortran 90, is still possible with external routines for which no INTERFACE block has been supplied. This only works because of assumptions made about how the data is stored.
(Note: When using external libraries like BLAS or LAPACK, this is still common practice. The performance improvement here is worth these 'dirty tricks' and well marked by the BLAS and LAPACK usage.)
- Try to avoid the use of `DATA` and `BLOCKDATA`. This functionality is now given by initializers.

5 Documentation

All large software systems have an enormous amount of associated documentation. Some remarkable part of the software process costs should be used to produce this documentation. Therefore, developer and management should pay as much attention to the documentation and its associated costs as to the development of the software itself.

The documentation associated with a system falls into two classes:

- Process Documentation
  These documents record the process of development and maintenance. Plans, schedules and project standards for example are process documentation.
- Product Documentation
  This documentation describes the product which is developed. System documentation describes the product from the point of view from the engineers developing and maintaining the system; user documentation provides a product description oriented towards system users.

5.1 Process Documentation

Process Documentation contains the following (important) features:

- Standards for coding rules, documentation, quality control, testing and verification.
  This document.
- For subsequent changes to the code: A version history and a changes log-file.
  The SCA distributes that with every version of the Software.
- Plans for future developments, milestones and deliverables.
  The Management maintains lists for these purposes.
- A list of known / reported bugs (Bug Tracker).
  This is maintained by the SCA.
- Regular reporting on the work done.
  The SCA of the COSMO Software are reporting to the SMC. This is done in the framework of the COSMO Priority Task *Support Activities*.

All process documentation will be available on the COSMO web page in the near future.

5.2 Product Documentation

Product Documentation may be split into two categories: external documentation (outside the code) and internal documentation (inside the code). In order for the documentation to be useful it needs to be both up to date and readable at centres other than that at which the code was originally produced. Since these other centres may wish or need to modify the imported code we specify that all documentation, both internal and external, must be available in English.

External Documentation

In most cases this will be provided at the package level, rather than for each individual routine. It must include the following:

- *Scientific Documentation*
  This sets out the problem being solved by the package and the scientific rationale for the solution method adopted. This documentation should be independent of (i.e. not refer to) the code itself.
- *Implementation Documentation*
  This documents a particular implementation of the solution method described in the scientific documentation and / or describes the structure of the code (for more technical oriented software). All public procedures (subroutines, functions, modules, etc.) in the package should be listed by name together with a brief description of what they do. A calling tree for routines within the package must be included. The main data structures must be described.
- *User Guide*
  This describes all things that are necessary to know for to use the software in an educated way. In particular, this describes in detail all inputs into the package. This includes both subroutine arguments to the package and any switches or 'tuneable' variables within the package. Where appropriate default values; sensible value ranges; etc should be given. Any files or namelists read should be described in detail.

The external documentation for the COSMO-Model and the INT2LM is written in LaTex. Other COSMO Software might define different formats.

Internal Documentation

This is to be applied at the individual routine level. There are four types of internal documentation, all of which must be present.

- *Procedure headers*
  Every subroutine, function, module, etc. must have a header. The purpose of the header is to describe the function of the routine, probably by referring to external documentation, and to document the variables used within the routine. All variables used within a routine must be declared in the header and commented as to their purpose including physical units if applicable. It is a requirement of this standard that the headers listed in Appendix A (or similar headers for different COSMO Software) be used and

completed fully. Centres are allowed to add extra sections to these headers if they so wish.

- *Section comments*
  These divide the code into numbered logical sections and may refer to the external documentation. These comments must be placed on their own lines at the start of the section they are defining. The recommended format for section comments is:
  ```
  !-----------------------------------
  ! <Section number> <Section title>
  !-----------------------------------
  ```
  where the text in <> is to be replaced appropriately.

- *Other comments*
  These are aimed at a programmer reading the code and are intended to simplify the task of understanding what is going on. These comments must be placed either immediately before or on the same line as the code they are commenting.

- *Meaningful names*
  Code is much more readable if meaningful words are used to construct variable and subprogram names.

All internal documentation has to be written in English.

5.3 Delivering New or Modified Code

If new or modified code is delivered, the programmer has also to provide the corresponding (external) documentation to the Editor of the Documentation System:

- Internal product documentation:
  The code has to be structured in meaningful units / sections, which are documented within the source code.

- External product documentation:
  For new code an external scientific (and implementation) documentation as well as an extension to the User Guide has to be provided. For modified code, the existing external documentation has to be updated.

- Process documentation:
  The programmer has to produce a (short) documentation of the changes to the existing software version that can be included to the version history and the changes log-file.

- All documentation has to be written in such a way that it can be taken over to the existing official documentation without substantial modifications or technical adjustments. It is planned to distribute the LaTex source files of the documentation together with the Source Code Repository.

6 Software Maintenance and Quality Control

The purpose of this section is to define a clear and transparent release management. This should ensure a proper implementation of new versions and a smooth updating of the operational systems. A necessary condition for that purpose is to give clear rules how the software can be updated.

6.1 Version and Release Management

The purpose of maintenance is that the product continues to meet the needs of the end-user. Over the lifetime of a system, its original requirements can be modified to reflect changing needs, the system's environment can change, and errors may show up. Therefore we can identify three categories of maintenance.

A Further developments:
If the requirements to the software are changing, the software must be extended to meet the new requirements. This means either that new components have to be developed or that existing components must be changed significantly in contents.

B Perfective maintenance (technical changes):
Perfective maintenance improves the system but does not change the functionality. These can be changes in parallelization, data I/O or also modifications to the general architecture of the software.

C Corrective maintenance (bug fixes):
This is needed to correct detected bugs and coding errors. Usually they are easy to correct but less easy to find sometimes.

Maintenance can be considered as an iteration of the development process and therefore the same standards and procedures should be applied. This means among other things:

- If necessary, new requirements must be formulated and validated.
  (E.g. the need for a new parameterization or a new dynamical core.)
- New components must be designed and implemented.
- If existing components of the system must be changed, there has to be a clear redesign before the implementation phase.
- Parts or all of the system must be tested and validated (Quality Control).

In short: there have to be similar rules to update the software and implement changes as there are for the initial development. For COSMO Software, Table 6.1 in Sect. 6.2 specifies these rules.

To keep track of all the changes, a version control system (VCS) must be used. With a VCS, the source code is stored at a central repository and the history of all files can be

recorded. Thus it is able to distinguish between the different versions of the software. The COSMO-Model and the INT2LM are maintained at DWD using a VCS, which is an in-house development, based on publicly available tools. `fieldextra` is maintained by MeteoSwiss using Subversion.

Every version which is not part of a VCS is called a private version and is available only for the developer. Only if code is provided to the SCA and checked into the VCS, it is part of an official version. For every official version the SCA defines a certain status to identify its availability for the users:

- Development Versions:
  These versions are intermediate versions that are only given to the developers for further work and testing.
  Such versions are necessary, if several changes by different developers have to be incorporated into an official version. With different development versions the SCA has the possibility to check in the different developments one by one.

- Test Versions:
  These versions are given to special test persons, who are conducting more extensive common tests.
  The individual changes have been tested separately by the developers. To check that the changes also work well in combination, further tests have to be performed. Also, different configurations have to be tested here.

- Released Versions (or Releases):
  If a version passes the common tests successfully, it can be released to all users.

For the COSMO-Model there are some flavors of releases, Depending on the different communities:

· NWP release: A version that passed the NWP tests (see 3.3, 3.4 in Table 6.1) and verification and is released to the NWP community.

· CLM release: A version that passed the CLM tests (see 3.5, 3.6 in Table 6.1) and standard evaluation and is released to the CLM community.
  NOTE: CLM subversions are maintained within the CLM community. Therefore, CLM releases may differ from any NWP versions of the code.

· Unified release: A model version which is a NWP and a CLM release.

For the COSMO-Model it is planned that each main model version (Version x.0) is a unified release for numerical weather prediction, regional climate modelling and environmental modelling.

The following sections are mainly written for the COSMO-Model and the INT2LM. Nevertheless, some of the rules are also applicable to other COSMO Software, but specific guidelines (e.g. for `fieldextra`) are given in the Appendices.

6.2 Rules for Implementation of Changes

All updates for the COSMO Software should follow two guidelines:

- Scientific
  The COSMO Software is further developed according to the COSMO Science Plan, which describes the COSMO goal and the strategy to attain the goal in the near future. Of course this does not mean, that developments can only find their way into the model, if they are reflected in the Science Plan. COSMO is still open for developments that do not (fully) adhere to this plan, if they are valuable for the community.
  Similar, but less extensive, plans do also exist for other COSMO Software.

- Technical:
  Implementation and documentation of all changes have to follow the *COSMO Standards for Source Code Development* (this document). This holds for all developments, whether they come from inside or outside of COSMO.

Depending on the categories of changes given in Sect. 6.1 and on the type of the code, the standards for updating software are different. For the COSMO-Model, changes of Category A (further developments) surely have to fulfill more requirements than technical changes. Table 6.1 below lists a set of rules, that have to be followed. Bug fixes and technical changes can skip some of these rules. The last columns indicate, which rules have to be obeyed.

Note that these rules apply to the COSMO-Model and the INT2LM. For other COSMO Software, similar rules may be specified.

Table 6.1: Rules for Implementation of Changes

No.	Description	Further Developments	Bug Fixes	Technical Changes
1	Information of Working Group Coordinators and discussion within the Management on planned changes / new developments. The Management maintains a list of actions for planned / ongoing developments, the *Planning for Future Developments.*	√		√
2.1	Design, development, implementation and documentation of the changes within COSMO or by external partners.	√		√
2.2	The *COSMO-Standards for Source Code Development* (this document) have to be followed. This is monitored by the TAG.	√	√	√

No.	Description	Further Developments	Bug Fixes	Technical Changes
3	Testing of the changes according to the Quality Control System of the COSMO partner or of the external partner that implements them. For NWP users this system must include at least the steps 3.1-3.4. For CLM users steps 3.1, 3.2, 3.5 and 3.6 apply.			
3.1 (all)	Pass the technical standard test suite (see Sect. 6.5). This is monitored by the SCA/TAG.	√	√	√
3.2 (all)	4-eyes-assurance: All changes must be checked by a second person. This is monitored by the SCA/TAG.	√	√	√
3.3 (NWP)	Testing of single cases with verification against observations.	√	√	√
3.4 (NWP)	If significant changes of results are expected, more intensive tests have to be performed, e.g. - Longer experiments, also for different seasons / domains / climates, with verification against observations. - Experiments to check interdependencies between data assimilation and forecast and also possible impacts on a fine-grid nest. (7 km → 2-3 km) - Additional experiments to check dependencies of results on the mesh-size (7 km as well as 2-3 km). - Verification results should be positive or neutral regarding the latest official release. (Note: The term "should" is chosen on purpose. Sometimes a change leads to a more physical formulation, although the results do not improve). (see Sect. 6.6)	√		
3.5 (CLM)	Individual tests exhibiting the effects of the modifications.	√	√	√
3.6 (CLM)	Standard evaluation of climate simulations.	√		
4	Presentation of the results: The results should be presented to the corresponding COSMO and / or CLM Working Group, if possible during the COSMO General Meeting, the COSMO User Seminar or the CLM Assembly, where appropriate.	√		

No.	Description	Further Developments	Bug Fixes	Technical Changes
5	Provide extensions / modifications for the Documentation System. Provide a test case and associated results, if appropriate.	√	√	√
6	For new components a code responsible person has to be nominated by the developer(s).	√		
7	Discussion within the Management about the implementation into the VCS and updating of the *Release Planning*. For COSMO, a final approval has to be done by the STC.	√		√
8	Submission of the changes to the SCA, who does a final check, whether the technical requirements (Coding Rules, Documentation) are met. He checks the modifications into the VCS and assigns a status (development, test, release) to the new version.	√	√	√

Some comments on these rules

1: Before the discussion in the SMC, new developments / changes can also be discussed in other groups (development teams, Working Groups, teams at universities, etc.), which want to contribute to the development of the COSMO Software. The SMC will then discuss and decide on the acceptance of changes and propose a Priority Project, a Priority Task or a Working Group Task to the STC.

2a: For major developments it is recommended to split up the design and implementation process in several steps to make sure early that scientific / technical requirements are met. This can be done by discussing the design with other scientists and / or the TAG.

2b: If developers are not sure about meeting technical requirements, the TAG should be contacted in an early phase of the development.

3a: Testing the changes:
Which tests have to be performed depends on the type of change. In the *list of actions*, the SMC should also specify for all developments, which tests are required.

3b: Bug fixes:
Bug fixes have to be implemented into an official version without much delay, but it has to be clear that it is a bug. This can be discussed with the Code Developer / Responsible and / or the Working Group Coordinator. The bug, the solution and the expected effects on the results have to be documented.

3c: Technical changes:
For technical changes no or only slight differences in the results are expected. Slight

differences due to numerical reasons (e.g. changing order of computations for optimization) are acceptable, but must be clearly documented.

3d: Verification:
Sometimes physically more consistent implementations can lead to worse verification scores (for what reasons ever). Changes can be taken over in such cases also.

3e: Timings:
For operational reasons, increasing run times are not desired. On the other hand it is obvious that more accurate / better algorithms can take more time to run.

6: In order to assure a good quality code and support in the future, there is the need that all components of the code do have a responsible contact person. This responsible person has to be affiliated at a COSMO member or a COSMO partner institution (institutions which are allowed to use the COSMO-Model). Preferably this should be the developer him- or herself, but alternatively, an institution responsibility can be accepted.

8: Before checking code into the VCS, the SCA will give the changes back to the developers for a last cross-check of the implementation.

Sharing of code with other software and incorporation of code from other communities

If code from the CLM- or the ART-communities should be incorporated into the COSMO-Model, the same rules do apply (e.g. the Unified Releases). The only exception is for code, which is only incorporated with `ifdef`, as is the case for e.g. COSMO-ART. Code that is belonging to the sources of the COSMO-Model (also the code between `ifdef` and `endif`) has to obey to the rules. Additional code, that is only linked afterwards (e.g. the sources from COSMO-ART), does not have to fulfill these rules.

There is also the necessity to incorporate code from other centres like e.g. the convection code from IFS. Such code is taken as it is to have it comparable to the IFS versions and to keep track of the changes there.

6.3 Rationale for Changes

All modifications of the COSMO-Model should follow the *Rationale for Changes*:

In the past we often had the situation that new features have been implemented into an official version as *optional*, even if they were not fully developed and tested yet. This approach ensures that the results of the operational applications are not affected, but leads to a rapid growth of the number of Namelist variables, with which the COSMO-Model can be controlled. Adding new features with this procedure more or less as a "technical change" must be avoided in the future. The Management not only has to decide about adding new features or variants of algorithms, but also needs to decide on which components can eventually be removed.

On the other hand there is a demand from the CLM community, that results should be reproducible between subsequent versions. Therefore we give the following recommendations when introducing changes:

- Bug fixes: Can be introduced without backwards compatibility.
- Technical changes: These should change the results only because of numerical reasons. If it can be shown by long term simulations that the changes are neutral, they can be introduced without backwards compatibility. If this cannot be shown, then these changes should be treated as the next item.
- Modifications which change the results, but should be adopted by everybody: Use a clearly documented internal switch in the source code. Once the change is commonly accepted, this switch can be removed again.
- Adding of new components: In this case a (new) namelist switch may be used.

6.4 Release Planning

In addition to the list of actions (*Planning for Future Developments*), the SMC will put up a roadmap for every COSMO Software (including priorities) for the upcoming changes (*Release Planning*). This includes both, new components as well as a planning for the removal of old or unused components and namelist switches. The planning should aim at having at most 2-3 major releases for the software every year. There might be more development and test versions. Especially bug fixes can be implemented in interim versions.

Release plans as well as other information about the source code management will be published on the COSMO web site. They have to be approved by the STC.

6.5 Standard Test Suite for the COSMO-Model

NOTE:
This section is work in progress. The TAG will complete it in collaboration with Working Group 6 until the next SMC Meeting in early 2012.

Depending on the category of changes, all contributions to the software have to go through the steps specified in Section 6.2 before they are submitted to the SCA. One aspect is to test pure technical issues like the independence of the processor configurations and similar things. For testing such issues, a standard technical test suite will be defined. The idea is to define such a test suite, that can easily be run at every center. Issues to be checked are for example:

- Portability
- Independence of processor configurations (MPI and OpenMP - for parallel code)
- Reproducibility of results with older versions
- Restart functionality
- I/O with Grib/NetCDF
- Tests with array bound checking
- Possibility to run with input data from different models (GME, IFS, ERA, etc.)

- Timings / efficiency / scalability

6.6 Meteorological Test Suite

NOTE:
This section is work in progress. The SMC will complete it for the NWP test suite until the General Meeting in 2012. The list of test cases as defined earlier by WGs 4 and 5 should be used as a starting point.

COSMO-CLM will complete the climate test suite in close collaboration with Ulrich Blahak until the General Meeting in 2012.

From here on we only give some ideas. This section has to be further elaborated

The Standard Test Suite is applied to two NWP and two CLM configurations and should check the above issues:

- NWP: 2 applications with different grid size based on the Runge-Kutta dynamical core
  - 7 km resolution (comparable to COSMO-EU)
  - 2.8 km resolution (comparable to COSMO-DE)
- CLM: 2 evaluation configurations over Europe
  - EU 0.44
  - EU 0.165

6.7 Testing the official versions in the VCS

If a contribution to the COSMO Software has fulfilled all steps specified above (especially: all the prescribed tests) and is accepted by the Management for a future official version, the code is given together with all documentation to the SCA. He incorporates them into the VCS according to the *Release Planning*. In accordance with the guiding authority (SMC for COSMO, CLM_CO and TAG for CLM) he assigns a status to each version: development, test, release. See Sect. 6.1 for the definition of these terms. The following tests will be done with the official versions:

- Development Versions:
  If several changes by different developers have to be incorporated into an official version, the SCA can build different development versions for that. The SCA checks the standard technical test suite for these versions.
- Test Versions:
  While the individual changes have been tested separately by the developers, the test versions (which summarize different contributions) are given to special test persons within the Consortium, who are conducting more extensive common tests. For the COSMO-Model, for example, DWD will test such versions in its "Parallelsuite".

- Released Versions (or Releases):
  If a test version passes these tests successfully, it can be released to all users.

7 Implementation Issues

This chapter should give some hints for efficient programming. All that follows is a draft and will be further developed in the near future.

NOTE:
This chapter is work in progress. The TAG will complete it until the SMC meeting in early 2012.

7.1 Memory Management

Allocation of Memory

The use of dynamic memory is highly desirable as it allows one set of compiled code to work for any specified resolution and allows the efficient reuse of work space memory. Care must be taken, however, as there is potential for inefficient memory usage, particularly in parallelized code. For example heap fragmentation can occur if space is allocated by a lower level routine and then not freed before control is passed back up the calling tree. There are three ways of obtaining dynamic memory in Fortran 90:

1. *Automatic arrays*
   These are arrays initially declared within a subprogram whose extents depend upon variables known at runtime e.g. variables passed into the subprogram via its argument list. Automatic arrays will not be retained once the subprogram ends.
2. *Pointer arrays*
   Array variables declared with the `POINTER` attribute may be allocated at run time by using the `ALLOCATE` command.
3. *Allocatable arrays*
   Array variables declared with the `ALLOCATABLE` attribute may be allocated at run time by using the `ALLOCATE` command. However, unlike pointers, allocatables were not allowed inside derived data types before Fortran2003 and some compilers might still not recognize it.

Space allocated using 2. and 3. above must be explicitly freed using the `DEALLOCATE` statement. In a given program unit do not repeatedly `ALLOCATE` space, `DEALLOCATE` it and then `ALLOCATE` a larger block of space. This will almost certainly generate large amounts of unusable memory and can degrade performance (depending on the platform). Using `ALLOCATE`s in every time step (or even more often) should therefore be avoided unless there is no other way to implement a special feature.

Always test the success of a dynamic memory allocation and deallocation. The `ALLOCATE` and `DEALLOCATE` statements have an optional argument to let you do this.

Passing arrays through argument lists

- Call-by-reference vs. Call-by-value

- Assume-shaped vs. explicit-shaped arrays

7.2 Working Precision

Use of the `KIND`*-type concept*

Fortran 90 provides representation methods for the intrinsic data types (`INTEGER, REAL, COMPLEX, CHARACTER, LOGICAL`). Each method can be specified by a value called *kind type parameter*, which is a scalar integer initialization expression that indicates

- the decimal exponent range for the integer type,
- the decimal precision and exponent range for the real and complex types
- and the representation methods for the character and logical types.

Each intrinsic type supports a specific set of kind type parameters. For `REAL`-types for example, every compiler must support the former types `single precision` and `double precision`. To specify the representation method, the kind type parameter can be given for some type specifiers (`INTEGER, REAL, COMPLEX`):

$$\texttt{REAL(KIND=}method_1\texttt{)} \text{ or } \texttt{REAL(KIND=}method_2\texttt{)}.$$

Here, $method_x$ are the kind type parameters. Intrinsic functions are provided to specify the kind parameter for a specified precision and variable range:

$$method_1 = \texttt{SELECTED_REAL_KIND (12,200)}.$$

specifies the representation method that has a working precision of at least 12 significant digits and an exponent range of at least 200. This is the `double precision` data type.

For `CHARACTER` data types, the *length type parameter* `LEN` specifies the number of characters in such an entitiy.

7.3 Parallelization

7.4 Optimization

7.5 Vectorization

7.6 I/O

7.7 Error Handling / Debug Messages

Just a comment:

"Error messages should be meaningful and easily understandable for the user and it should be possible to find them in the source code. It is not sufficient just to give an error code number; especially it should be avoided to compose this number as a sum of an index with some basis number (which makes it too difficult to locate the error with a `grep` or similar)."

References

[1] Andrews, Philip (UKMO), Gerard Cats (KNMI/HIRLAM), David Dent (ECMWF), Michael Gertz (DWD), and Jean Louis Ricard (MeteoFrance), European Standards For Writing and Documenting Exchangeable Fortran 90 Code, Version 1.1, 1995.

[2] E. Kalnay, et al., Rules for Interchange of Physical Parametrizations, *Bull. A.M.S.* 70, No. 6, p. 620, 1989.

[3] Sommerville Ian, Software Engineering, Fourth Edition, Addison-Wesley, 1992.

A Standard Headers for the COSMO-Model and INT2LM

The standard headers are presented in this appendix. They are written as templates. Text inside < > brackets must be replaced with appropriate text by the user.

Program Header

```
!+ <A one line description of this program>
!----------------------------------------------------------------------

PROGRAM <NameOfProgram>

!----------------------------------------------------------------------
! Description:
!   <Say what this program does>
!
! Method:
!   <Say how it does it: include references to external documentation>
!   <If this routine is divided into sections, be brief here,
!        and put Method comments at the start of each section>
!----------------------------------------------------------------------
!
! Input files:  <Describe these, and say in which routine they are read>
! Output files: <Describe these, and say in which routine they are written>
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
! Language:           Fortran 90.
! Software Standards: COSMO Standards for Source Code Development
!----------------------------------------------------------------------

! Modules used:

USE, ONLY : &
! Imported Type Definitions:
! Imported Parameters:
! Imported Scalar Variables with intent (in / out):
! Imported Array Variables with intent (in / out):
! Imported Routines:
!
! <Repeat from USE for each module...>
!----------------------------------------------------------------------

IMPLICIT NONE

!----------------------------------------------------------------------
!
! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable
!
! Local parameters / scalars / arrays:
!
!- End of header ------------------------------------------------------

!----------------------------------------------------------------------
! Program Body
!----------------------------------------------------------------------

END PROGRAM <ProgramName>
```

Subroutine header

```
!+ <A one line description of this subroutine>
!----------------------------------------------------------------------

SUBROUTINE <SubroutineName> (<InputArguments, inoutArguments, OutputArguments>)

!----------------------------------------------------------------------
! Description:
!   <Say what this routine does>
!
! Method:
!   <Say how it does it: include references to external documentation>
!   <If this routine is divided into sections, be brief here,
!        and put Method comments at the start of each section>
!----------------------------------------------------------------------
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
! Language:           Fortran 90.
! Software Standards: COSMO Standards for Source Code Development
!----------------------------------------------------------------------

! Declarations:
! Modules used:

USE, ONLY : &
! Imported Type Definitions:
! Imported Parameters:
! Imported Scalar Variables with intent (in / out):
! Imported Array Variables with intent (in / out):
! Imported Routines:
!
! <Repeat from USE for each module...>
!----------------------------------------------------------------------

IMPLICIT NONE

!----------------------------------------------------------------------
!
! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable
!
! Subroutine arguments
! Scalar arguments with intent(in):
! Array  arguments with intent(in):
! Scalar arguments with intent(inout):
! Array  arguments with intent(inout):
! Scalar arguments with intent(out):
! Array  arguments with intent(out):
! Local parameters:
! Local scalars:
! Local arrays:
!
!- End of header ------------------------------------------------------

!----------------------------------------------------------------------
! Subroutine Body
!----------------------------------------------------------------------

END SUBROUTINE <SubroutineName>
```

Function header

```
!+ <A one line description of this function>
!----------------------------------------------------------------------

<Type> FUNCTION <FunctionName> (<InputArguments>)

!----------------------------------------------------------------------
! Description:
!   <Say what this function does>
!
! Method:
!   <Say how it does it: include references to external documentation>
!   <If this routine is divided into sections, be brief here,
!        and put Method comments at the start of each section>
!----------------------------------------------------------------------
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
! Language:           Fortran 90.
! Software Standards: COSMO Standards for Source Code Development
!----------------------------------------------------------------------
!
! Declarations:
! Modules used:
!
USE, ONLY : &
! Imported Type Definitions:
! Imported Parameters:
! Imported Scalar Variables with intent (in):
! Imported Scalar Variables with intent (out):
! Imported Array Variables with intent (in):
! Imported Array Variables with intent (out):
! Imported Routines:
!
! <Repeat from USE for each module...>
!----------------------------------------------------------------------

IMPLICIT NONE

!----------------------------------------------------------------------
! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable
!
! Function arguments
! Scalar arguments with intent(in):
! Array  arguments with intent(in):
! Local parameters:
! Local scalars:
! Local arrays:
!
!- End of header ------------------------------------------------------

!----------------------------------------------------------------------
! Function Body
!----------------------------------------------------------------------

END FUNCTION <FunctionName>
```

Module header

```
!+ <A one line description of this module>
!----------------------------------------------------------------------

MODULE <ModuleName>

!----------------------------------------------------------------------
! Description:
!   <Say what this module is for>
!----------------------------------------------------------------------
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
!
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
! Language:           Fortran 90.
! Software Standards: COSMO Standards for Source Code Development
!----------------------------------------------------------------------
!
! Modules used:
!
USE, ONLY : &
! Imported Type Definitions:
! Imported Parameters:
! Imported Scalar Variables with intent (in):
! Imported Scalar Variables with intent (out):
! Imported Array Variables with intent (in):
! Imported Array Variables with intent (out):
! Imported Routines:
!
! <Repeat from USE for each module...>
!
! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable
!----------------------------------------------------------------------

IMPLICIT NONE

!----------------------------------------------------------------------
! Global (i.e. public) Declarations:
! Global Type Definitions:
! Global Parameters:
! Global Scalars:
! Global Arrays:
!
! Local (i.e. private) Declarations:
! Local Type Definitions:
! Local Parameters:
! Local Scalars:
! Local Arrays:
!
! Operator definitions:
!   Define new operators or overload existing ones.
!----------------------------------------------------------------------

CONTAINS

!----------------------------------------------------------------------
! Define procedures contained in this module.
!----------------------------------------------------------------------

END MODULE <ModuleName>
```

B Amendments for fieldextra

Style and Typing Rules

A set of additional coding rules and conventions are defined in the file README.developer distributed with the fieldextra package. At the time of the writing, this document is a work in progress based on the release 10.4 of fieldextra.

Note that most of the rules defined in Chap. 4 of this document apply. The most notable exceptions are:

- DIMENSION:
  Use of DIMENSION statement is recommended.
- USE modulename, ONLY:
  Is not required for general purpose modules, such as modules used for definition of program entities (fxtr_definition) or modules used to manage program diagnostic (support_information).
- GOTO:
  Usage is accepted when dealing with error conditions to jump to the error processing part of the procedure.
- Variable names:
  Do not use the naming conventions associated with the variable type (e.g. LOGICAL starts with l and similar).
- Multiple statements on the same line:
  Is allowed for short statements logically related (e.g. exception handling, initialization of some local variables).

Documentation

The documentation for fieldextra is written in plain ASCII.

The following documents are available within the fieldextra package or on the COSMO web site:

- CREDITS: list of contributors;
- HISTORY: version history and change log files, list of known bugs;
- ROADMAP: plans, milestones and deliverables;
- README.install: installation guide;
- FirstContact.pdf, README.user, README.user.locale and FAQ: user guide;
- README.developer and README.cosmo_api: implementation guide.

Software Maintenance and Quality Control

`fieldextra` is maintained by MeteoSwiss using Subversion.

The code design is described in the documentation of `fieldextra`.

A test case with its associated results should be provided with any new development.

All changes to the code have to be tested using the standard set of examples provided within the fieldextra package.