IMAGE Programming Guidelines

Background Studies

Netherlands Environmental Assessment Agency

IMAGE Programming Guidelines

Programming requirements and guidelines

PBL (E. Stehfest, L. de Waal),
VORtech

Netherlands Environmental Assessment Agency

IMAGE Programming Guidelines

PBL publication number 500110007

Corresponding Author: Liesbeth.deWaal@pbl.nl; IMAGE-info@pbl.nl

This publication can be downloaded from our website: www.pbl.nl/en. A hard copy may be ordered from: reports@pbl.nl, citing the PBL publication number.

The Netherlands Environmental Assessment Agency (PBL) is the national institute for strategic policy analysis in the field of environment, nature and spatial planning. We contribute to improving the quality of political and administrative decision-making by conducting outlook studies, analyses and evaluations in which an integrated approach is considered paramount. Policy relevance is the prime concern in all our studies. We conduct solicited and unsolicited research that is both independent and always scientifically sound.

Office Bilthoven
PO Box 303
3720 AH Bilthoven
The Netherlands
Telephone: +31 (0) 30 274 274 5
Fax: +31 (0) 30 274 44 79

Office The Hague
PO Box 30314
2500 GH The Hague
The Netherlands
Telephone: +31 (0) 70 328 8700
Fax: +31 (0) 70 328 8799

E: info@pbl.nl
www.pbl.nl/en

Rapport in het kort

IMAGE Programming Standard

Dit rapport beschrijft de eisen en richtlijnen die gesteld worden aan de programmatuur van het IMAGE model. De aanleiding was onder andere een fundamentele herstructurering van de programmatuur, voor IMAGE versie 2.5. De in dit rapport gestelde eisen en richtlijnen hebben betrekking op zowel ontwerpoverwegingen als op onderhoud- en overdraagbaarheidaspecten. De ontwerpoverwegingen geven richtlijnen over onderwerpen zoals programmastructuur, modelhiërarchie, het gebruik van data modules en het foutmeldingensysteem. Onderhoud- en overdraagbaarheidaspecten hebben betrekking op richtlijnen over bijvoorbeeld het Fortran 90 standaard, naamconventies, code lay-out, interne documentatie.

Trefwoorden:
IMAGE, programmeerstandaard, eisen, richtlijnen, onderhoud.

Contents

Summary

This document describes the requirements and guidelines for the software of the IMAGE program. The motivation for this report was a substantial restructuring of the source code for IMAGE version 2.5. The requirements and guidelines relate to design considerations as well as to aspects of maintainability and portability. The design considerations determine guidelines about subjects, such as program structure, model hierarchy, the use of data modules, and the error message system. Maintainability and portability aspects determine the guidelines on, for example, the Fortran 90 standard, naming conventions, code lay-out, and internal documentation.

Keywords:
IMAGE; programming; guidelines; requirements; maintenance

1 Introduction

1.1 The IMAGE framework and the IMAGE program

IMAGE is an ecological-environmental modelling framework that simulates the environmental consequences of human activities worldwide. It represents interactions between society, the biosphere and the climate system to assess sustainability issues, such as climate change, biodiversity and human well-being. The objective of the IMAGE framework is to explore the long-term dynamics of global change as the result of interacting demographic, technological, economic, social, cultural and political factors.

A central part of the IMAGE framework – and subject of this report – is the IMAGE program (executable) that contains all land- and climate-related models.

One of the reasons for this report was the substantial restructuring of the source code for IMAGE version 2.5. Therefore, these programming guidelines are applicable for IMAGE version 2.5 onwards, reflect the situation in 2009, and will be updated as required.

1.2 Aim of the report

The IMAGE program is very complex. First of all, this is due to the large number of interacting models within the program. Secondly, many of these models are themselves complex by nature. Thirdly, IMAGE works on very large data-structures (arrays) which easily leads to excessive memory use if the programming is not done efficiently. Finally, the code is under constant development to include new insights, which causes program-fragments of different development stages to coexist within the program.

To keep the IMAGE program code manageable, a software structure has been established and coding guidelines have been developed. This report describes the software structure and the coding guidelines.

Note that the rules and guidelines in this report are not meant to hinder developers, but rather to ensure the overall integrity of the IMAGE program. If not carefully observed, the IMAGE code may become unmanageable, over time. Therefore, IMAGE developers are kindly but strongly requested to please comply with the guidelines. If necessary, guidelines can, of course, be changed after thorough consideration.

1.3 Contents of this report

This section provides an outline, to give the reader a quick impression of the report's contents. This report contains six chapters.

Chapter 2, Program structure, discusses the various aspects of the software structure and the underlying reasons for it. This chapter describes how IMAGE, as a whole, is assembled from individual models. Each model is programmed in such a way that it can, in principle, be replaced by another implementation without affecting the other parts of IMAGE. This modularity is very important for keeping the large number of models manageable.

Chapter 3,Files used by the IMAGE program, discusses the main types of files that are used by the IMAGE program.

Chapter 4, The restart , discusses the requirements of the program code that are imposed by the restart mechanism. The restart mechanism allows for a simulation to start off with data produced by a previous simulation. The restart functionality has strong implications for the structure of the code, and must therefore be particularly well understood by the programr.

Chapter 5, Calibration functionality, discusses the implementation of the calibration process within the IMAGE code.

Chapter 6, Writing output messages, discusses the routines that should be followed to produce error messages and warnings.

Chapter 7, Code aspects, discusses the guidelines that keep the IMAGE code portable, readable and consistent. It treats the following subjects:

Naming conventions (Section 7.2)
In this section, guidelines are given, for example, on naming files, subroutines, and modules.

Coding guidelines (Section 7.3)
This section gives rules for the source code of the system.

1.4 Other relevant documents

In this report, we also refer to the following reports and documents, in which additional information can be found:

- IMAGE User Manual (PBL, 2010)
- Description of the calibration functionality (IMAGEcalibration.doc)
- Description of the IMAGE version control (IMAGE_versionControl.doc)

Links to these documents can be found at the IMAGE project directory, under \Image_Intro\RelevantLinks, or they can be obtained, if relevant, from image-info@pbl.nl.

The IMAGE project directory is '\project\M481508_IMAGEaanpassingEnBeheer', at the moment.

2 Program structure

2.1 Overview

The IMAGE program consists of:
1. models, such as AOS, TES, and ATMOCHEM., each consisting of a number of subroutines, and
2. libraries which contain general subroutines used throughout the program, for example, to read and write files.

These models require elaborate explanations and are the topic of Sections 2.2 through 2.5. Libraries are discussed in Section 2.7.

2.2 Models within IMAGE

2.2.1 Introduction

The IMAGE program consists of a hierarchy of models. The two main models are the Atmosphere Ocean System (AOS) and the Terrestrial Environment System (TES). Both consist of underlying models. For example, AOS consists of models, such as ATMOCHEM, CLIMATE, and OCEAN, which may, in turn, consist of yet other models.

The models are organised in a hierarchy to make the code easily accessible: at the main program level, only the two main systems AOS and TES are visible. Someone interested in the AOS model can zoom-in on that model without being concerned with the internal structure of TES. This helps a programmer to quickly find his/her way around the code.

Also, the hierarchical structure (with additional requirements on how models are programmed, see the next section) makes it possible to replace the entire AOS model with an entirely different implementation without affecting the rest of the IMAGE program. Within AOS, the same arguments hold for the models from which AOS is composed, and subsequently also hold for all models down the line, up to the deepest models within the IMAGE program.

The interaction between same-level models (e.g. AOS and TES) takes place through variables that are passed as arguments to the calls to AOS routines and TES routines. The two models do not use each other's variables directly. Hence, the variables in the calls to the AOS and TES routines represent the complete interaction between the two models. This ensures that one of the models can be replaced by a different model implementation which provides the same variables. Also, it makes the interactions between models explicitly visible, which helps to understand the model as a whole (In fact, there is one extra line of interaction between models on the same level in the hierarchy, which is through the grid files; as will be discussed further on in this report).

Each model, at each level in the hierarchy, consists of the same elements: for example, each has an initialisation routine and a routine to perform a single time step. Understanding this structure is important for several reasons. First of all, a programmer familiar with this structure can quickly find his/her way around the code. Secondly (but not less important), the structure is essential for the restart mechanism, which will not work if the structure is compromised. Finally, the structure guarantees that the interaction between models within the IMAGE program is always through subroutine arguments only, which in turn guarantees that each model can be replaced by another or new implementation and helps to keep the interactions visible and understandable.

2.2.2 Outline of the structure of a model

A structure of a model is schematically shown in the Figure 2.1:

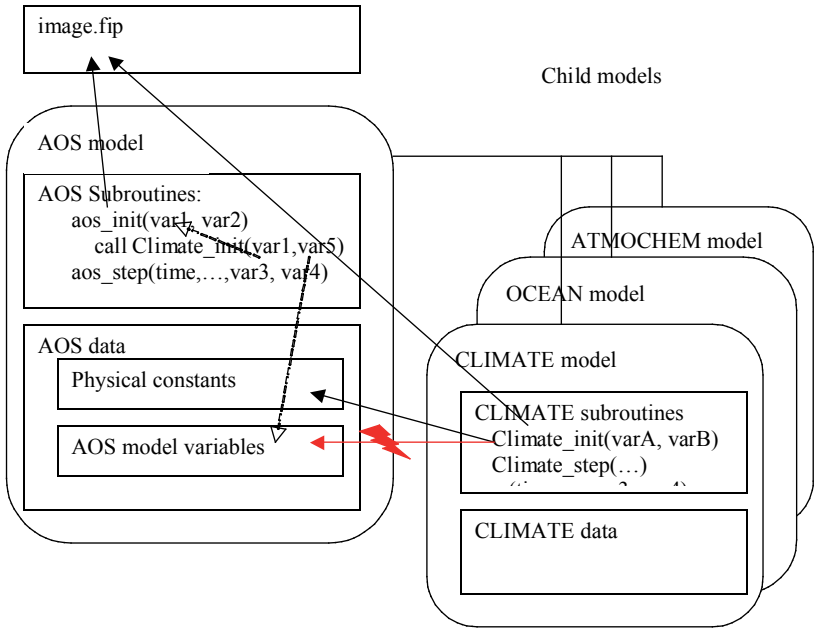
On the lower left side is the AOS model. It consists of two parts: 1) a number of subroutines and 2) a collection of data, which is implemented as a Fortran MODULE.

The set of subroutines consists of standard subroutines which every model has to have (such as an initialisation routine and a time step routine) and some extra subroutines that are specific to the AOS model. The latter category consists of subroutines that are used to keep the standard subroutines short and clear: these extra subroutines are always called from the standard subroutines.

The data associated with the IMAGE model also consists of two parts: a collection of (physical) constants and a collection of model state variables. The constants are set once at the start of the program and are never changed. The model state variables represent the state of the AOS model, which obviously varies with (simulation) time.

On the right side of the figure above are the child models of AOS (e.g. CLIMATE, OCEAN, ATMOCHEM). Each of these child models has a structure that is identical to that of AOS.

Figuur 2.1



There are some very important rules about passing data from a model to its child models:

- Subroutines of a certain model may directly use the physical constants in the parent model (so climate_init can directly use physical constants from the AOS data module).
- Subroutines of a certain model may directly use the state variables of their own model.
- Subroutines from a model may *not* directly use the model state variables from the parent model and, reversely, the parent model may *not* directly use model state variables from one of its child models. If a model variable from AOS is needed in CLIMATE, then it must be passed as an argument in the call to the climate routine (var5 in the figure above). Technically, however, it is possible to make use of state variables in the forbidden way, because the child model will often include the _DataDecl module from the parent model to access the physical parameters. In such a case, it could also access the state variables of the parent. To avoid this, the keyword ONLY must be used after the USE statement of the parents _DataDecl module, and there may only be physical constants after the ONLY statement. When accepting a new piece of code as part of the operational version, this should always be checked.
- Variables that are input in the subroutines of AOS can obviously also be used in the call to subroutines of child models that are called from that AOS ('passing through'), such as var1 in the figure above.

The rationale behind this is that it forces the programmer to explicitly show the interactions between a model and its child models (and from one child model through the parent model to another child model). This, in turn, guarantees that a model can always be replaced by another implementation. Also, it allows for easier understanding of the interaction between the models and, thus, for a better understanding of the model as a whole.

Actually, it would be better if access to the physical constants of the parent model was also not allowed, but this would lead to excessive subroutine argument lists.

The image.fip file, shown at the top of the figure above, contains constants (Fortran: PARAMETER) that are used throughout the program, such as the number of regions and the number of cells. The image.fip file can be included at any point in the code and the constants can be used freely. As with the physical constants, this breach of strict modularity is allowed to avoid excessively long argument lists.

The next section summarises the rules for defining new models within the IMAGE program hierarchy and the rules for access to data.

2.3 Guidelines for models

2.3.1 Where to place a new model

When a new model is added to the IMAGE program, the following rules are applicable, to preserve the structure, efficiency, understandability, and maintainability of the program:

- *Hierarchy:* A new model must always be placed at a logical location within the hierarchy, in the same way as existing similar models.
- *Ownership:* A new model must belong either to one, ,and only one other model, or to the main program. In other words, there must be *aparent-child relationship*.
- *Input/output:* The model should be placed under a parent model in which a large part of the input and output variables of the new model are already present. The number of extra I/O variables with higher level models must be minimal.

- *Coherence:* model subroutines may only be called by other subroutines within that model or by subroutines of the parent model.
- *Source file directories and Visual Fortran:* The source file directory structure and the Visual Fortran project file structure must match the model structure.

2.3.2 What defines a new model

Models must have the following characteristics.

- *Significance:* A model must be meaningful. That is, it must represent a model as understood by a modeller.
- *Data:* The number of variables that is exchanged across model boundaries must be as small as possible, to support the clarity and maintainability of the program.
- *Size:* The size of a model must be limited. If a model is large, the possibility of dividing it into smaller models should be examined, to improve the structure and understandability of the program.

2.3.3 Interaction between models

The following rules apply to data flow.

- *Data exchange:* All data exchanged *between models* is either passed through subroutine arguments or by means of grid files, but not through common blocks or data modules. For more information about grid files, see Chapter 3.
- *Model state variables:* Model state variables (the variables that are defined in the _DataDecl module) may only be accessed by the subroutines of the model itself.
- *Physical constants:* Constants of a model module may be accessed by subroutines of this and lower level models.

These rules are aimed at keeping data as local as possible, so that adjustments made to a model have a limited impact.

2.4 Model data module

Every model has a *data module*:

Module <model name>_DataDecl

This Fortran module contains the model variables that must retain their value after a model subroutine finishes. In principle, the module does not contain subroutines.
The module contains two types of data: *physical constants* and *state variables*. The *physical constants* are in fact variables that are kept unchanged during the simulation run. They are dynamically initialised in subroutine *<model name>_DataInit* by reading the values from a *.const* file. See also Chapter 3 for more information on *.const* files.

The *modelstate variables* are statically initialised by the module itself and are modified by the model subroutines.

2.5 Standard model subroutines

Every model has a number of *standard subroutines*; these are described in the following subsections.

<model name>_DataInit

This subroutine initialises the model's physical constants. These are initialised by reading the appropriate values from the *<model name>.const* input file, by means of general subroutine *getini*. For more information on the *<model name>.const* file, see Chapter 3.

Example: see *aos_DataInit.f90*

<model name>_init

This subroutine calls *<model name>_DataInit*, and the *init* routines from the child models (for an explanation about child models, see Section 2.2.2). Furthermore, it sets the model state variables to values matching the start of the simulation run.

Note: The *init* routines of the highest level models (AOS and TES) are called from the main program unit *(image.f)*.

Example: see *ocean_init.f90*

<model name>_step

This subroutine computes one time step for the model. This routine calls the *step* routine of each child model, if any (see Section 2.2.2 for an explanation about child models). It also performs the calculations for the model itself.

Note: For a description of the tasks that the *step*-routine has to perform for the restart functionality, see Chapter 4.

Note: The *step* subroutine of every highest level model (AOS and TES) are called from the main program unit *(image.f)*.

Example: see *climate_step.f90*

<model name>_DataExport

This subroutine exports the model state variables to a *Data restart file*. For a further description of this routine, see Chapter 4.

Example: see *aos_DataExport.f90*

<model name>_DataImport

This subroutine imports the model state variables from a *Data restart file*. For a further description of this routine, see Chapter 4.

Example: see *aos_DataImport.f90*

<model name>_CallvarsExport

This subroutine exports the value of subroutine arguments to a *Callvars restart file*. For a further description of this routine, see Chapter 4.

Example: see *aos_CallvarsExport.f90*

<model name>_CallvarsImport

This subroutine imports the values of subroutine arguments from a *Callvars restart file*. For a further description of this routine, see Chapter 4.

Example: see *aos_CallvarsImport.f90*

Copyfunf_<model name>
This subroutine copies restart files of child models to the working directory. For a further description of this routine, see Chapter 4.

Example: see *copyfunf_ldm.f90*

2.6 Other model subroutines

Next to the standard model subroutines described in the previous section, a model may also contain additional subroutines. They are mostly called from _init or _step, in order to prevent that _init and _step become too long. Additional subroutines in a model can make use of all model state variables declared in its _DataDecl. The additional subroutines may also call each other. In fact, the programmer has much freedom in how to use the additional subroutines, as long as the initialisation functionality remains in (or under) the _init routine, the time stepping functionality remains in (or under) the _step routine and state variables remain in the _DataDecl module.

If the content of a separate subroutine is a large and clearly defined functionality, it should be considered to cast as a new model. In this case, the separate subroutine will often be the _step routine of the new child model and an appropriate _DataDecl module (probably containing some of the model state variables that were previously in the parent model) and accompanying _init routine must be created.

To determine whether a separate subroutine is actually a child model, the _DataDecl module that would be created for the child model can be analyzed. If there is nothing to be put in the _DataDecl module, then the separate subroutine is apparently a stateless routine and does not qualify as a model. If, however, the _DataDecl module can be put to good use, then this is a clear indication that the separate subroutine is part of a child model. See also Section 2.3.2 for characteristics that every model should have.

2.7 Libraries

The program currently has one library, called gentools. It contains a large number of supporting subroutines, often embedded in modules. These subroutines comprise general functionality that is not specific for any model. New subroutines for this library should contain the same kind of functionality.

At this moment, it is possible to include ('use <module name>') the following modules from library GENTOOLS:

name of module	description
calibration	functionality to run IMAGE in calibration mode, see Chapter 5
funf_mod	functionality to read and write unformatted grid files, see Section 3.3
getinifile	functionality to read const files, see Section 3.4
gettables	functionality to read tables from dat-files
imexport	functionality to import or export data to restart files, see Section 3.5
lintpol_mod	linear interpolation functionality
message	contains the message system, see Chapter 6
mfmake_mod	provides an interface to mfmake routines, see Section 3.2

3 Files used by the IMAGE program

3.1 Introduction

IMAGE uses the following main types of files (file formats are explained in the appendix):

- *Regional input/output files*, providing values of a variable for each region, for a number of years.
- *Grid files*, which mainly provide a mechanism to pass variables that are defined on the cell level from one model to another, without the need to keep all these variables simultaneously in memory (which requires a large computer memory). Additionally, grid files are also used for input and output of gridded information.
- *Const files*, which provide the values of physical constants used by models. There is one const file for each model.
- *Restart files*, which store the state of all state variables of Image at a certain moment in the simulation, so that the simulation can be resumed from that moment on.

This chapter presents a brief discussion on each of these files.

3.2 Regional files

A regional file stores a regional input or output variable.

3.2.1 Writing regional output files

A regional output file is created using the subroutine *mfmake*. This subroutine creates the file and writes the first lines (the header of the file). This is typically done in the *<model name>_init* routine of a model, which makes all necessary preparations for the first time step of the model.

Adding lines to a regional output file is done by calling the subroutine *mfappend* as illustrated by the following code fragment:

```
      call mfappend(88,'LOCCROPPR',time)
      do c=1,NFC
         do l=1,NLATIT
            write(88,9005)(val(r,c,l),r=1,NRT)
         enddo
      enddo
      call mfclose(88,time)
9005  format(<NRT>(1X,F6.4))
```

This fragment adds a section to the file LOCCROPPR.OUT. The section starts with the value of time. In this case, the section has NFC times NLATIT lines of NRT values each.

The *mfclose* operation writes the end-of-file marker (a ']') at the end of the file, if time is the last time step of the simulation; otherwise it does nothing but close the file.

3.2.2 Reading regional input files

Regional input files are read as illustrated by the following code fragment:

```
      call mfopen(88,'INEMN2O.OUT')
      ttaben2oin =  0
1010  continue
      call ryear(88,year,endflag)
      if (endflag .eq. 0)then
         ttaben2oin =  ttaben2oin+1
         taben2oin(1,ttaben2oin) =  year
         read(88,*)(taben2oin(1+r,ttaben2oin),r=1,NR)
         goto 1010
      endif
      close(88)
```

The *mfopen* subroutine opens the INEMN2O.OUT file for reading and scans over the first lines with header information. The call to *ryear* determines whether there is data for at least one more year to be read and sets *endflag* accordingly. If so (i.e. *endflag.eq.0*), then the values are read. After the data from the last year has been read, the file is closed with a normal *close* operation.

3.3 Grid files

Within the IMAGE framework, grid files are mainly used to pass array contents directly from one program to another or from one subroutine to another. Grid files are unformatted Fortran direct access files. Every grid file contains one array for a single moment in the simulation; the name of the array appears in the filename, for example. greg.unf1 (for array greg).

The advantage of these files is that data can be exchanged while keeping memory space within acceptable limits. Grid

files are, in general, not supposed to last after the simulation has been completed.

There are a few exceptions:
- Several grid files are provided as input for the simulation:
  - *<project>\data\unf* (files coming from several sources)
  - *<project>\data\gcm* (global climate model data)
  - *<project>\start* (terrestrial vegetation data)
- Several grid files contain output of the simulation, in particular the files in
  - *<project>\<scenario>\map* (scenario output files)
- There are also grid files among the restart files. These are actually copies of grid files that are used during the simulation to pass an array from one model to another.

Grid files are only accessed by the subroutines *readfunf* and *writefunf* from module *funf_mod* in library *gentools* section *file*.

Technical details about grid files can be found in *Appendix B*.

3.4 Const files

Every model has an associated *.const* file containing the values of the physical constants used by the model. Having constants initialised from a control file, instead of hard-coded constants, makes it possible to experiment with sensitivities. Furthermore, in case of maintenance, physical constants only have to be adapted at one location.

The values in *const* files are only accessed using the general subroutine *getini*.

3.5 Restart files

The restart files are placed in folders; one folder per model. The folders' structure matches the model hierarchy. The restart grid files are located in

<project>\<scenario>\work\state.

Restart files come in three different flavours:
- Callvars files, which contain the values of the arguments in the call to the step routine of the model for a particular time step.
- Data files, which contain the values of the model state variables at a particular moment during the simulation.
- Grid files, which are copies of the grid files produced at a particular moment during the simulation by the model to pass large arrays (with the size of the number of cells) to other models.

The Data and Callvars files can be either textual or binary. The grid restart files are always binary (unformatted). For a description of the regular textual file format, see the IMAGE User Manual, Appendix A File Structures, Section Restart Files.

There is a Data restart file and a Callvars restart file for every model.

For more information about the restart functionality, see Chapter 4.

The restart functionality

4

4.1 Introduction

An important feature of the IMAGE system is the restart functionality. The restart functionality comprises three main functions:

1. *Restart*: the possibility to start an IMAGE run at an arbitrary year for which the state of all the models has been exported in a previous IMAGE run.
2. *Nocompute*: the possibility to ignore the functionality of certain selected models in a simulation run.
3. *Readfromfile*: the possibility to let certain selected models, that have the *nocompute* option set, read their results from files produced by a previous run.

The action of producing files for use in later runs is called 'exporting'; a run 'exports' files; the created files are called 'export files'. The action of reading export files is called 'importing'; a run can 'import' files.

The user aspects of the restart functionality are described in the IMAGE User Manual. There are two situations in which a programmer must pay attention to keep the restart functionality working:

1. when making changes to an existing model
2. when adding a new model to the IMAGE program

The next two sections, in general terms, describe what a programmer must do to keep the restart functionality working in either of these situations. The routines that are mentioned in these sections are described in more detail in Sections 4.4 to 4.7. Background information about the various aspects and functions of the restart functionality is given in Appendix A.

IMPORTANT: The subroutine descriptions in this section must be considered as basic descriptions. In practice, variations and deviations in details are possible. See existing models as examples, for instance, AOS, TES, and TVM.

4.2 Restart aspects when making changes to an existing IMAGE model

For a restart, the state of the model at the moment of restarting must be restored. The state of the model is determined by:

1. The arguments in the call to the <*model name*>_*step*-routine.
   These values are read and written by means of routines <*model name*>_*CallvarsImport* and <*model name*>_*CallvarsExport*. When the arguments in the call to the *step* routine are changed, these two routines must be changed accordingly. The programmer must pay attention to the import and export routines always corresponding with each other.
2. The module variables that are declared in the <*model name*>_*DataDecl* (see Section 2.4).
   These values are read and written by means of routines <*model name*>_*DataImport* and <*model name*>_*DataExport*. When changes are made to the <*model name*>_*DataDecl*, these two routines must be changed as well. Again, the programmer must take care that the import and export routines always correspond to each other.
3. The variables stored in grid files (see Section 3.3).
   Grid files are used to pass array contents directly from one subroutine to another. Grid files are unformatted Fortran direct-acces files and are only accessed by subroutines *readfunf* and *writefunf* from the module *funf_mod*. If these routines are used without optional arguments, the grid data is read from or written to the working directory. When a run is restarted, the information stored in the working directory is no longer available. If a model needs data from a grid file written in a previous time step, the programmer must take care that this information is written to the restart directory, as well.
   To do this, the optional arguments *exportYear* and *sysInfoNr* are available. If these arguments are included in the call to *writefunf*, the grid file is written to the working directory as usual, and also to the restart directory, with a time label attached to the grid file. The data can be restored from the restart directory with *readfunf* in combination with the parameters *exportYear* and *sysInfoNr*.

4.3 Restart aspects when adding a new model to IMAGE

When a new model is added to the IMAGE program, the programmer must make sure that:

1. The routines <*model name*>_*init* and <*modenamel*>_*step* of the new model contain certain code fragments, see Sections 4.4 and 4.5.
2. The state of the model, that is, the calling arguments, the model state variables declared in the <*model name*>_*DataDecl* and (if used) the unformatted grid files, are imported and exported by means of the five standard subroutines described in Section 4.6. The import and

export routines must always correspond to each other: the same number of variables must be read to and written from file, in the same order.
3. Extra keywords are added to the file *imexport.dat*, see Section 4.7 for more details.

4.4 The model init routine

The model init routine *<model name>_init* should contain the following blocks. The logic of these blocks is described here in pseudo code.

Checking the nocompute option (at the start of the code)
if *nocompute* is true then return to the calling routine

Making export files (at the end of the code)
call *<model name>_DataExport* for the current year

4.5 The model step routine

The model step routine *<model name>_step* should contain a number of blocks related to the restart functionality. The logic of these blocks is described here in pseudo code.

The first three blocks appear at the start of the executable code:

Checking nocompute and readfromfile
if *nocompute* is true then
if *readfromfile* is true then
call *<model>_CallvarsImport* for the current year
call *copyfunf_<model name>*
GOTO block "*Export grid restart files*" below
else
RETURN to the calling routine

Note: The statements 'call copyfunf_<model name>' and 'GOTO block "Export grid restart files" below' are only present when the model uses grid files.

Note: The call to *copyfunf* is made because in case of *nocompute* the model does not call its child models. This call ensures that data from restart grid files is also read for the child models.

Note: Instead of a call to *copyfunf_<model name>*, a list of calls to copyfunf routines of child models may be issued.

Note: Instead of this block, the call to *copyfunf_<model name>* may also be issued in the block '*Export grid restart files*' below.

Check if run is restart run
if time is restartyear then
call <model name>_DataInit
call <model name>_DataImport for the preceding year (time-1)

Import grid restart files (only when the model uses grid data)
for every grid array for which a restart file is available do
call *readfunf* with *exportyear* =F *time* (=F current year).

Note: By calling *readfunf* with the argument *exportyear=time*, the subroutine checks if the current year (*time*) is the restart year. If so, it reads the grid array from the corresponding restart grid file; otherwise, it reads the array from the intermediate grid file. For arrays that must always be read from the intermediate grid file, *readfunf* should be called without the argument *exportyear*.

The following blocks appear at the end of the executable code:

Export Data and Callvars restart files
call <model name>_DataExport
call <model name>_CallvarsExport

Note: These subroutines write model state variables and the values of the arguments in the call to the step routine to restart files when the current year (time) is an export year.

Export grid restart files (only when the model uses grid data)
for every grid array for which a restart file must be written do
call *writefunf* with *exportyear* = *time* (= current year).

Note: By calling *writefunf* with the argument *exportyear=time*, the subroutine checks if the current year (*time*) is an export year. If so, it writes the grid array to the restart grid file; in any case, it writes the array to the intermediate grid file. For arrays that must only be written to the intermediate grid file, *writefunf* should be called without the argument *exportyear*.

Examples: see *Appendix E*, *Code lay-out examples*.

4.6 Standard model subroutines for restart

This section broadly describes the way in which the model's standard subroutines for restart work, in pseudo code. See also Section 2.5.

Subroutine <model name>_CallvarsImport
This subroutine reads the values of the arguments in the call to the step-routine from the 'Callvars' restart file.
call *OpenImport*
for every subroutine header variable do
call *DataImport*
call *CloseImport*

Subroutine <model name>_DataImport
This subroutine reads model state variables from the 'Data' restart file:
call *OpenImport*
for every state variable do
call *DataImport*
call *CloseImport*

Subroutine <model name>_DataExport

This subroutine writes model state variables to the 'Data' restart file:

```
if year equals exportyear then
    call OpenExport
    for every state variable do
      call DataExport
    call CloseExport
```

Subroutine <model name>_CallvarsExport

This subroutine writes the value of the arguments in the call to the step routine to the 'Callvars' restart file:

```
if year equals exportyear then
    call OpenExport
    for every variable do
      call DataExport
    call CloseExport
```

Subroutine copyfunf_<model name>

This subroutine copies restart files of child models to the working directory:

```
for every child model do
    call copyfunf_<child_model>
```

Examples: see *Appendix E, Code lay-out examples*.

4.7 Control file imexport.dat

The control file *imexport.dat* contains keywords that control the way the restart functionality works. For a description, see the Image User Manual.

When a new model is added to the program, the keywords *<model name>_nocompute* and *<model name>_readfromfile* must be added to *imexport.dat*, if these options are desired for the new model.

Examples: see the file *imexport.dat* itself.

5 Calibration functionality

This chapter describes the implementation of the calibration functionality in IMAGE. Detailed information of the calibration process itself can be found in the IMAGEcalibration.doc file (see Section 1.4).

The code for the calibration process is incorporated in the main IMAGE code: all calibration runs are performed with the same executable as used for the production runs. The advantage of incorporating the calibration code in the main IMAGE code is that it is much easier to maintian: modifications that are made to the main IMAGE code are automatically made to the calibration code, as well. In this way, the chances that the main IMAGE code and the calibration code diverge from each other, unintentionally, are very small.

The IMAGE program is run in calibration mode by setting the appropriate calibration run number (hereafter referred to as *runno*) in the file *calibration.dat* in the *dat* directory (for production use of IMAGE, *runno* is -1). The calibration process consists of a number of runs that each perform a subset of the full IMAGE model for different time periods. The restart functionality, described in Chapter 4, and in particular the *nocompute* function (control file *imexport.dat*) is used to bypass complete models that are not needed in a specific calibration run. Table 1 presents an overview of which models are used in the different calibration runs, as well as the time period and time step for each run.

Sometimes, the calibration code is meant to be different from the main IMAGE code. For this reason, a logical function *incalibrun* is available through the module *calibration*. This function only returns .TRUE. if one of the arguments is equal to *runno*. Modifications within a model or extra output of the calibration process are incorporated in the code by means of an if construct *if (incalibrun(runno))*. As an example, the following lines taken from *tes_step*, where subroutine *migrate* must not be called in calibration runs 5, 6 and 7:

```
if (time.eq.timestart) then
    if (time .ne. MINYEAR) then
        call adm70(tfeede,tfeedi,pop,agrproda,agrtradea, &
            agrtradec,dssra,dssrc,nrane,nrani,optlist)
        call lcm70(arprod,croparea,frothc,frharv,grazintens,
        mf)
        call lrm70(arprod,agrproda,agrtradea,agrtradec,dssra,
        dssrc)
    endif
    if (.not.(incalibrun(5,6,7))) call migrate
    (time,optlist%migropt)
endif
```

If the modification is just a few lines, it can be put directly under the if construct. Larger modifications or modifications that cannot be incorporated in the same file are to be put in extra files. At this moment the following extra files exist:

- tes_step_run4, performs a modified tes_step
- adm_calib_run6, performs extra computations in model *adm* for run6 and run7
- lcm_step_run8, performs a modified lcm_step
- tes_step_run9, performs a modified tes_step
- copy_calib_run10, performs extra copying action for a large number of files
- copy_calib_run12, performs extra copying action for a large number of files

The calibration process is to be performed in the directory IMAGE25. The main batch file is *PREPROCESSING_1_13.bat* in the directory bat. This script runs the complete calibration process. In the init*data* directory of IMAGE25 there are files *calibration_runno.dat*, *imexport_runno.dat* and *time_runno.dat* and some more run specific files present with the settings for each calibration run. These files are copied to the *runxx/dat* directory before a specific calibration run. For each calibration run, a batch script *mkrunXX.bat* (dir:bat/mkrun) is called that performs these copying actions and other necessary preparations.

For a description of the whole calibration process, see also the IMAGEcalibration.doc on the calibration directory.

Routines calculated in the different calibration runs

Tabel 5.1

	run1	run2	run3	run4	run5	run6	run7	run8	run9	run 10	run 11	run 12
AOS[1]	x	x	x	x	x					x	x	x
Atmochem	x	x	x	x	x					x	x	x
addemiss												
Climate												
Ocean												
Radiat												
Readies												
Slr												
Worldtozoom												
TES	x	x	x	x	x	x	x	x	x	x	x	x
Admlcm				x		x	x	x	x	x	x	x
Adm						x	x	x	x	x	x	x
Lcm				x		x	x	x	x	x	x	x
wood				x				x	x	x	x	x
Drivforce						x	x	x	x	x	x	x
Biofuel												
Ccm				x						x	x	x
Carbon				x						x	x	x
Luem												x
Soil	x	x	x	x	x			x	x	x	x	x
Tvm	x	x	x		x					x[2]	x[3]	x[4]
Wat		x	x							x	x	x
Writetes									x	x	x	x
Period												
Timestart	1970	1970	1765	1765	1970	1970	1970	1970	1970	1970	1970	1970
Timestep	1	1	5	1	5	1	1	1	5	1	1	1
Timestop	1969	1969	1970	1969	2000	2000	2000	2000	2000	2000	2000	2000

[1] Only atmochem_init is called for initialisation reasons, AOS is then set to nocompute.
[2] Retrun from tvm_init immediately after climscale with tvm set to nocompute
[3] Retrun from tvm_init immediately after climscale with tvm set to nocompute
[4] Retrun from tvm_init immediately after climscale with tvm set to nocompute

6 Writing output messages

Messages must be written using the message system (see *message.f90*).

To activate the message system, the subroutine *readMsgSettings* must be called once, at the start of the IMAGE program, so before any calls to message subroutines. The subroutine *msg_stop* is called once at the end of the IMAGE program to stop the message system after the last call to a message subroutine.

The options in the file *message.dat* determine the amount of output that is generated by the message routines. For example, an option can be set to turn off any message that is not an error message. In this case, calls to subroutines to produce, for example, warnings or information will not produce any output.

Producing a message requires two steps in the code. First, the program has to prepare the message text in the character buffer array *msgstr* stored in the module *message*, and then call the message subroutine:

- write the message into *msgstr*
- call <*type*>*msg* with *msgstr* as argument, where <*type*> is *error*, *warn*, *info*, *trace* or *debug*:
  - *errormsg*
    This subroutine produces an error message. This type of message is given when an undesired situation is detected that has a serious impact on the results from the program. In many cases, proceeding has no use and the program will stop after issuing the error message. Error messages cannot be suppressed by settings in the *message.dat* file.
  - *warnmsg*
    This subroutine produces a warning message. This type of message is given when an undesired situation is detected that has little or no effect on the results from the program.
  - *infomsg*
    This subroutine produces an informative message. This type of message is given to provide the user with some information about the process.
  - *tracemsg*
    This subroutine produces a trace message. Every subroutine issues a trace message before starting its actual processing. These messages give information about the location of an error if one should occur. Every subroutine must start with a call to *tracemsg*, see Section 7.4.1.
  - *debugmsg*
    This subroutine produces a debug message. Calls to *debugmsg* are placed at specific places within the program process. Debug messages present intermediate results, which give the developer extra information about the process that is useful while searching for the location of a program malfunction.

All the <*type*>*msg* routines have an optional second argument (after *msgstr*) that allows the programmer to assign a level to the message. This level indicates how serious the message is. If the level is high, then this indicates that the message is serious; a low level indicates a message that is of less interest. By setting a threshold level in the *message.dat* file, the user can indicate how serious a message must be to be actually printed or written to screen. Messages with a level below the threshold in *message.dat* are ignored by the message system. In this way, the user can determine the amount of output that is produced.

7 Code aspects

7.1 Terminology

1. *Parameters:* Fortran PARAMETER constants, such as array dimensions and loop limits.
2. *Physical constants:* the 'constants' in the _DataDecl module (which are actually variables that are given a value only once at initialisation). See Section 2.2.2 for an explanation.
3. *Model state variables:* Variables shared by routines of one model (as opposed to local variables within subroutines, that can only be accessed from the subroutine itself) and which are defined in the _DataDecl module. See Section 2.2.2 for an explanation.
4. *Header variables:* Subroutine arguments.
5. *Module:* A Fortran MODULE, that is, a module as defined in the Fortran programming language.
6. *Model:* A set of subroutines and a module that together implement a mathematical description of a particular aspect of the environment. See Section 2.2 for a discussion of the concept of models in IMAGE.
7. *Subroutine:* A Fortran SUBROUTINE, that is, a subroutine as defined in the Fortran programming language.

7.2 Naming conventions

1. *Subroutine and module names:* names are of the form <model name>_<function>, where <model name> is the name of the model, and <function> is composed of acronyms beginning with an upper-case letter. The file containing the subroutine or module has the same name.
   Examples: 'aos_DataDecl', 'ccm_DataInit'.
2. *PARAMETERs:* names are in upper case.
   Note: No underscores are used.
   Examples: 'maxyear', 'realsize'.
3. *Physical constants (in the models DataDecl file):* names are in upper case.
   Note: No underscores are used.
   Examples: 'tabwdistag', 'ercat1'.
4. *Variables (in the models DataDecl file and in subroutines):* names are in lower case. Upper case letters to distinguish acronyms or component names are allowed.
   Note: No underscores are used.
   Examples: 'eco2ab', 'ForestManag', 'frNH3spread'.
5. *Variables defined on cell level as global arrays (grid variables):* as all variables, except that the name starts with a 'g' for global. However, when subroutines only handle one grid cell (e.g. fnpp), the variables still start with a 'g'.
   Examples: 'gnep', 'gfrac'

7.3 Coding guidelines

1. *Programming language:* The programming language is Fortran 90.
2. *Source code format:* New code must be written in Fortran 90 free format. The source code file extension is *.f90*.
3. *Source files:* In principle, each subroutine is contained in a separate source file. Exceptions are subroutine modules in libraries.
4. *Modules:* Modules may only be used for model data modules '*_DataDecl*' (one per model) and subroutine modules in libraries.
5. *Subroutine length:* A subroutine should not contain more than 600 lines, comment lines included.
6. *The save statement:* Save statements are not allowed outside model data modules. Variables that must not become undefined should be stored in the module. Otherwise, they cannot be reached by the import and export functions for restart purposes.
7. *Real type declaration:* Variables and arrays of type single precision real must only be declared by the keyword real, and not by real*4. Otherwise, the length of real variables cannot be controlled by a compiler option; see Appendix D.
   Likewise, variables and arrays of type double precision real must only be declared by the keyword double precision, and not by real*8.
8. *PARAMETER constants:* Parameter constants should only be declared in the file *image.fip*. These constants are of a general, not model-bound nature. Constants that are specific for a model should be declared as physical constants in the model data module.
9. *Local subroutine variables:* Local subroutine variables must be dynamically initialised, with assignment statements, and not in the declaration _DataDecl at the model level.
10. *Hard-coded numbers:* The source code should not contain any hard-coded numbers. These numbers should be defined either as Fortran PARAMETERs or as physical constants in the _DataDecl module.
11. *Tabs:* Tabs are not allowed in the source code, because editor-dependent lay-out must be avoided. The indentation depth must be three spaces per level.
12. *Message system:* Messages must be written using the message system. See Chapter 6.
13. *Code comment:* Comment in new code always starts with an exclamation mark ('!'). When the comment is on a separate line, the exclamation mark is in the first position.

14. *Format statements:* Format statements of one program unit must be grouped together, and form the last statements before the *end subroutine* statement.
15. *Use statement:* Unless all variables of a module are used in a subprogram, the variables which are actually used should be selected by means of the keyword only.
16. *Do construct:* In new code, only the block form of a do construct must be used. This avoids having to use statement labels.
17. *Common blocks:* The use of common blocks is prohibited.

7.4 Code lay-out

7.4.1 Subroutine lay-out

Every subroutine has a standard lay-out, as follows:

- A *comment heading* containing four parts which must contain meaningful text:
  - The subroutine name.
  - The 'called by' statement
  - The 'objective' statement
  - The section with information from the version of the management system
- The *first block of statements*, in a fixed order:
  - Subroutine statement
  - Use statements
  - Implicit 'none' statement
  - Comment header with text 'Input/output variables'
  - Input and output variable declarations
  - Comment header with text 'Local variables'
  - Local variable declarations
  - Comment header with text 'Initialisation'
  - Initialisation statements
  - Comment header with text 'Start of source code'
  - Statements concerning nocompute and readfromfile (only in step routines)
  - Statements concerning restart run (only in step routines)
  - A call to *tracemsg*
- The *actual processing code*
- The *last block of statements*
  - Calls to export subroutines (only in step routines)
  - Comment header with text 'Formats'
  - Format statements
  - The *end subroutine* statement

For an *example* of the subroutine lay-out, see Appendix F, Code lay-out examples.

7.4.2 Data module lay-out

Every data module ('_DataDecl') has a standard lay-out, as follows:

- A *comment heading* containing four parts:
  - The module name.
  - The 'called by' statement (empty)
  - The 'objective' statement.
  - The section with information from the version of the management system
- The *block of statements*, in a fixed order:
  - Module statement
  - Use statements
  - Implicit 'none' statement
  - Save statement
  - Comment header with text 'Exported constants'
  - Exported constants
  - Comment header with text 'Exported variables'
  - Exported variables (with initialisations)
  - End module statement

Note: The term 'exported' is used outside the context of the restart functionality. Here it means 'exporting' items *from* the module *to* the subroutines.

For an *example* of the module lay-out, see Appendix F, Code lay-out examples.

7.4.3 Lay-out of MODULES other than <model name>_DataDecl

The code contains several Fortran90 MODULES other than the <model name>_DataDecl modules. The message MODULE is a good example. In fact, the use of MODULES is encouraged, as this is the preferred way of programming in Fortran90.

The lay-out of such a module is as follows:

- A *comment heading* containing four parts:
  - The module name.
  - The called-by section. (empty)
  - The purpose section.
  - The subversion section
- The *block of statements*, in a fixed order:
  - Module statement
  - Use statements
  - implicit 'none' statement
  - Exported constants
  - Exported variables (with initialisations)
  - Exported procedures
  - Internal constants
  - Internal variables
  - Internal procedures
  - 'contains' statement
  - subroutines and functions
  - End module statement

Every block is preceded by an appriopriate comment header.

Examples: see *funf_mod.f90* and *message.f90*.

8

Version Control

At the moment, svn and tortoise are used as version control tools for development of the IMAGE model. All standard functionalities, such as creating tags and branches, are available. The IMAGE developer is expected to use this version control, the most recent install files and User manual are provided (see Section 1.4).

9 Checklist IMAGE Programming Standard

This checklist is meant as an assisting tool for inspecting the IMAGE code. Inspection happens when changes have been made, or when a new model has been developed.

The table below shows a number of topics on which the program can be examined. When the inspection of one topic is finished, a checkmark can be placed in either the column 'Yes' or the column 'No'. When in the 'No' column, a number can be denoted referring to the text below the checklist describing the nature of the problem.

Tabel 9.1

Inspection topic	Yes	No
In case of a new model, are all requirements related to the model relations met? See Section 2.3.1.		
Does the new model conform to the model definition requirements? See Section 2.3.2.		
In case of a new model, does the new code obey the rules concerning the data flow? See Section 2.3.3.		
In case of (an addition to) a library, does the new code meet the requirements for library items? See Section 2.7.		
In case of a new model, do the standard data module and subroutines exist, and do they conform to the requirements? See Sections 2.4 and 2.5.		
In case of a new grid file or a new regional file, does the new file obey the rules for these files? See Chapter 3.		
Do new or modified step routines and the standard subroutines conform to the rules relating to the restart functionality? In case of new model, are the appropriate keywords added to the file imexport.dat. See Chapter 4.		
In the new or modified code, are messages written with the message system in the appropriate way? See Chapter 6.		
Does new source code obey the rules for naming conventions and coding guidelines? See Sections 7.2 and 7.3.		
In case of a new source code file (e.g. subroutine, module, library) , does it obey the rules for code lay-out? See Section 7.4.		

Descriptions of findings

1. Description 1.
2. Description 2.
3. ….

Appendix A
Restart functionality

This appendix gives an overview of the various aspects concerning the restart functionality. It is meant as background information to better understand the rules imposed on models when using this functionality. See Section 4.

Functions

The restart functionality supports three functions:

- *the restart run*
  The restart functionality enables the program to resume a simulation at a particular year for which the state of the entire model has been saved in a previous IMAGE run.
- *the nocompute option*
  The restart functionality offers each model the possibility to skip its own computations.
- *the readfromfile option*
  The restart functionality provides every model with the possibility to read data from an earlier run and use it as computation results instead of performing the calculations itself.

Types of restart files

There are three types of restart files:

- Data restart file : contains model module variables
- Callvars restart file : contains model header variables
- Grid restart file : this is a copy of a grid file as used during the simulation

Initialisation of the variables that control a restart

The control variables used by the restart functionality (e.g. the year in which the restart run should start) are stored in the Fortran 90 module *ImExport*. This module contains data and a set of supporting subroutines. The data is initialised by subroutine *InitalizeImExport*, which is called by the main program unit of IMAGE.

Restart run

When *restartyear* in the file *imexport.dat* is non-zero, the simulation run is a restart run. A restart run starts off with data from the restart year, written by a previous run. The restart data is contained in Data restart files, and also in grid files. The main program unit sets the starting time to *restartyear* and every model will obtain its starting data from restart files.

Note: A restart run needs data *from the year before the restart year*. So, when for instance a restart run is started in the year 1995, it needs starting data from the year 1994.
Note: The restart run does *not* read the *physical constants* from the restart file, but gets them as usual from the appropriate *.const* files.

Nocompute option

For every model there is a *nocompute* keyword in the file *imexport.dat*. When this keyword is 1, the model does not perform any computations.

Readfromfile option

For each model there is a keyword *readfromfile* in the file *imexport.dat*. When *nocompute* is true (no computations are performed) and *readfromfile* is true, the model reads data from a Callvars restart file, and sometimes also from grid files. The imported data will be used as computational results.
Note: The readfromfile model(s) will need restart files for all time steps concerned.

Exporting restart files

Restart files are created and filled (i.e. 'exported') for every year that is an export year. Export years are defined in the control file *imexport.dat*, with the keyword *expyears*.
Note: Exported restart files support restart runs as well as readfromfile runs.

Appendix B
Grid file technical details

Format

Gridfiles are unformatted Fortran direct access files. Each gridfile contains only one record (see Fortran manuals for a description of the concept of records: records are the units into which Fortran files are divided), which contain the stored array. The record length (recl) is equal to the number of bytes occupied by the array. A compiler option (assume:byterecl) ensures that the record length of unformatted files can be given in bytes.

File name

The grid file name is constructed from several elements, as follows:

G<filnam>[_<year>][.<dimlen>].[<gridtyp>]UNF<arrtyp>[.R<expyear>]

The square brackets ([…]) indicate that the enclosed part may be empty.

Clarification of the name elements:

G	This letter is always the first character of the file name, indicating that it is a global array.
<filnam>	The base name of the file, referring to the array name.
_<year>	The simulation year to which the array relates. This year may appear in both input and output file names.
<dimlen>	The length of the second array dimension. This length appears when the array is two-dimensional. The length of the first dimension is equal to the number of grid cells.
.	The separation character between the file name and the extension.
<gridtyp>	The grid type of the array: <empty> means: land, and eventually inland water cells 'T' means: grid-cell number indexed by row and column (row, column) 'F' means: total grid cell area, uniform for each latitude, therefore, dimension row
UNF	The fixed name part of the extension; it means 'unformatted'.
<arrtyp>	The array type: 0 means type real 1 means type integer*1 (or logical*1) 2 means type integer*2 4 means type integer*4
.R<expyear>	The simulation year of the export file. This year appears when the file is an export file for restart.

Examples: *greg.unfo*, *gfrac_1985.19.unfo*, *gdaytmp.12.unfo.r1975*.

The size of a grid file with a real array depends on the number of array elements, and on the size of each array element. The latter is related to the precision of real arrays, see also Appendix D.

Appendix C
Regional Files

Regional files are text files that contain values over a number of years, for all regions. The values can be scalars (one value per region), 1D arrays (a list of values per region), 2D arrays (a 2D array per region) or 3D arrays (a 3D array per region).

Regional files are used both as input files (scenario files with extension '.scn' or otherwise with extension '.out') or output files (with extension '.out')

The structure of regional files is as follows:

The first line has the following structure (in Extended BNF notation):

"!",("real"|"integer"),<varname>,"
["<dimname>{,<dimname>},"]",[" (t)"],
"; Unit = ",<unitname>,"; Label = ",<label>

with the following explanation:

!	indicates that this is a comment line
real|integer	indicates whether values are reals or integers
<varname>	the name of the variable that is stored in the file
<dimname>	name of a dimension, for example, NAPT or NR. The last dimension is usually NR or one of its derivatives, such as NRT
'(t)'	is only present if the variable is time dependent
<unitname>	name of the units in which the variable is given, for example, Gg/yr
<label>	a text to explain the meaning of the variable.

Example:
! real AGRTRADEA [NAPT,NRT] (t); Unit=F Gg/yr; Label=F Net export of animal products

The second line has the following structure (in Extended BNF notation):
"!",("real"|"integer"),<varname>," ["<dim>{,<dim>},"]",["
(t)"]," = ["

With the following explanation:

real|integer	indicates whether values are reals or integers
<varname>	the name of the variable that is stored in the file
<dim>[,<dim>*]	the numerical value of each dimension, for example, '6,25'
'(t)'	is only present if the variable is time dependent

Example:
real AGRTRADEA [6,25] (t) = [

After the first two lines, number of blocks follow. Each block consists of a first line that only contains the year for which the block gives the values (e.g. '1970'), which is followed by the values of the variable for that year. The number of values is obviously the product of all dimension sizes. Each line has a number of values that is equal to the last dimension (typically NR or one of its derivatives). Successive lines concern successive indices in the one-but-last dimension, and so on.

The last line of the file only holds the character '];' to indicate the end of the file.

Appendix D
Real arrays and grid files

In IMAGE, each real array is declared with the type declarant real. Normally, these arrays have single precision. However, if desired, the precision of these arrays can easily be changed by submitting the compile time option */real_size*:

- */real_size* = 32 means single precision (occupies 32 bits)
- */real_size* = 64 means double precision (occupies 64 bits)

In Visual Fortran, this option can be set by the path:

Project ⇨ Settings ⇨ Fortran ⇨ Default Real Kind:

- Default Real Kind = 4 means single precision (occupies 4 bytes)
- Default Real Kind = 8 means double precision (occupies 8 bytes)

Note: When setting the *compiler option/real_size*, the *program parameter constantrealsize* (in *image.fip*) must be set accordingly:

- realsize = 4 means single precision (occupies 4 bytes)
- realsize = 8 means double precision (occupies 8 bytes)

The precision of real arrays in *input grid files* is defined by the programs that write those files. The IMAGE program has to know the actual precision; therefore, this is defined by the parameter constant input_realsize (in *image.fip*):

- input_realsize = 4 means single precision (occupies 4 bytes)
- input_realsize = 8 means double precision (occupies 8 bytes)

The IMAGE defaults are:

- Internal real arrays have single precision; parameter constant realsize is 4; compiler option *real_size* is 32.
- *Note*: This implicates that real arrays in intermediate grid files also have single precision.
- Real arrays in input grid files have single precision; parameter constant input_realsize =F 4.
- *Note*: after reading, these arrays are converted to internal arrays.

See also Appendix E, Software, Program generation, option *real_size*.

Appendix E Software

Development environment

Microsoft Visual Studio / C++ vs 6.0
Compaq Visual Fortran Standard Edition vs 6.6C
SubVersion vs 1.4.5 (freeware)

Program generation

The production version of IMAGE is generated by 'Win32 Release' as active configuration. The more important compiler options are mentioned here; the values shown are examples. For a detailed enumeration of all options, see the file *IMAGE_PC.dsp*.

Relevant options:

/architecture:k7	The code is generated for processor type *AMD Athlon*.
/tune:k7	The generated code is optimised for processor type *AMD Athlon*.
/assume:byterecl	The record length of unformatted files is expressed in bytes.
/optimize:0	No optimisation is applied.
/real_size:64	The precision of real variables is set to double precision.
/automatic	This option requires that local variables be put on the run-time stack.
/stack:0xffffffff	In relation to the option */automatic*, this link option assigns a large size to the run-time stack.
/fpconstant	This option requires that a single-precision constant assigned to a double-precision variable be evaluated in double precision.

System dependencies

At some points, the source code contains deviations from the *Fortran90 standard*. In principle, this may cause problems on other platforms.

- Type declarations with the asterisk character (*) to indicate size. Examples: integer*1, integer*4, integer*8, logical*1. (Note that real*4 and real*8 are not allowed, see Section 7.3)
- The use of a variable between angle brackets in a format statement. Examples: format(<NRT>(1x,ES24.15e3)), format(<NCHLOR>(1x,ES24.15e3)).

Appendix F
Code lay-out examples

Subroutine lay-out

```
!===============================================================================

! Subroutine: SLR_STEP

!===============================================================================

! Called by : AOS_STEP

!===============================================================================

! Purpose  : Calculate sealevel time step

!===============================================================================

! Version info:
    !  $URL: file:///V:/…… /src/aos/slr/slr_step.f90 $
    !  $Revision: 303 $
    !  $Date: 2007-09-28 12:54:01 +0200 (vr, 28 sep 2007) $

!===============================================================================

subroutine slr_step(time, climdat, dtem, tocn)

use constants, only: NOCLAY, NSEAL, NSEALT
! subroutine modules
use message,  only: tracemsg
use imexport
! imported types
use aos_DataDecl, only: Climatedattype
! data modules
use slr_DataDecl
use timeSettings,  only: timestart

implicit none

!===============================================================================

! Input/output variables
real, intent(in) :: time
type(Climatedattype),  intent(in) :: climdat
real, intent(in) :: dtem
real, dimension(NOCLAY), intent(in) :: tocn

!===============================================================================
```

```
! Local variables
integer :: oclay
integer :: glac

real  :: tl1
real  :: coefe
real  :: sibit
real  :: si

! Sea level rise (cm)
real, dimension(NSEALT) :: sealevel =F 0.

!=======================================================================

! Initialisation
oclay  =F 0
glac  =F 0

tl1 =F 0.
coefe  =F 0.
sibit  =F 0.
si =F 0.

sealevel =F 0.

!=======================================================================

! Start of source-code

! Check if this system's computations need to be done
if (sysinfo(SS_AOS_SLR)%no_compute) then
    if (sysinfo(SS_AOS_SLR)%read_from_file) then
            call slr_CallvarsImport(time, climdat, dtem, tocn)
    endif
    return
endif

! Check if a restart is wanted
if (restart(nint(time))) then
    call slr_DataImport(time-1)
endif

call tracemsg('slr_step')

<actual processing code>

call slr_DataExport(time)
call slr_CallvarsExport(time, climdat, dtem, tocn)

!=======================================================================

! Formats
9 format(<NSEALT>(1x,ES24.15e3))

end subroutine slr_step

!=======================================================================
```

Data module lay-out

```
!===============================================================================

! Module : RADIAT_DATADECL

!===============================================================================

! Called by : -

!===============================================================================

! Purpose : This module contains shared definition data for RADIAT

!===============================================================================

! Version info:
    !  $URL: file:///V:/M02a_schemas_mnp/svn-repos/... /radiat_DataDecl.f90 $
    !  $Revision: 303 $
    !  $Date: 2007-09-28 12:54:01 +0200 (vr, 28 sep 2007) $

!===============================================================================

module radiat_DataDecl

use constants, only: NR, FRITER, NITER
use aos_DataDecl, only: Radiativeforcingtype

implicit none

!---------------------!

! EXPORTED CONSTANTS!

!---------------------!

! Indicator for regional conversion from IMAGE regions to regions of
! Climate research Group (neccessary for downscaling of climate) ARRAY
integer, dimension(NR,FRITER:NITER), save :: RDOWN

! Tropospheric ozone sensitivity coefficient (W/m2/DU). Table 6.3 TAR
real, save :: TROPOZSENS

! Radiative forcing coefficients (W/m2*ppb)
! Data from IPCC WG1 TAR Government and expert review draft Table 6.7

! Chlorides
real, save :: RADCFC11
real, save :: RADCFC12
real, save :: RADCFC113

! Halons
real, save :: RADHA1211
real, save :: RADHA1301
real, save :: RADCH3BR

! PFCs
real, save :: RADCF4
real, save :: RADC2F6
real, save :: RADSF6
```

```
! HFCs
real, save :: RADHFC23
real, save :: RADHFC32
real, save :: RADHFC4310

!-------------------!

! EXPORTED VARIABLES!

!-------------------!

! Radiative forcings in 1990 (W/m2)

type(Radiativeforcingtype),  save :: qrf90

! Scaling factor for reference year (1990; as in Schlesinger, 2000)
real, dimension(FRITER:NITER), save :: gscale =F 0.

end module radiat_DataDecl
```

Colophon

Responsibility
Netherlands Environmental Assessment Agency (PBL)

Authors
PBL (E. Stehfest, L. de Waal), VORtech

Lay out
Studio RIVM

Contact
Liesbeth. deWaal@pbl.nl
IMAGE-info@pbl.nl

IMAGE Programming Guidelines

This document describes the requirements and guidelines for the software of the IMAGE system. The motivation for this report was a substantial restructuring of the source code for IMAGE version 2.5. The requirements and guidelines relate to design considerations as well as to aspects of maintainability and portability. The design considerations determine guidelines about subjects, such as program structure, model hierarchy, the use of data modules, and the error message system. Maintainability and portability aspects determine the guidelines on, for example, the Fortran 90 standard, naming conventions, code lay-out, and internal documentation.

Netherlands Environmental Assessment Agency, June 2010