1.4.2 Major data structures

The spectral fields are carried in the module YOMSP, in which the arrays SPA3 and SPA2 hold the 3D and 2D state variable spectral fields. Individual fields within these arrays are addressed via pointers which are defined in the same module. The grid point fields have a much more flexible storage structure, which was introduced at cycle 27, and is designed to allow the easy incorporation of new prognostic variables, without the need to know about and modify a large number of routines through IFS. The basic concept is that all the grid point variables are stored within a single structure, and that any routine which performs a generic operation on grid-point data just loops over all the grid point fields within the structure. There is, however, the potential to control action for individual fields by the use of a set of *attributes* which are associated with each field in the structure.

There are two core data-structures:

GMV Contains prognostic variables involved in the semi-implicit (u, v, T, p_s in the hydrostatic model). This can be considered to be a "fixed" data structure, with little reason for modification. The prognostic fields all have a spectral representation, and can be either two or three dimensional. There are no attributes, apart from field pointers, associated with the GMV fields.

GFL Contains all the other variables (currently q, q^l, q^i, a, O_3 in ECMWF's operational model). This is a more flexible structure that can be easily extended. All the fields are three dimensional, with the vertical extent always the number of levels in the model. The fields may have a spectral representation or be pure grid-point fields. A number of attributes are available to govern the treatment of the field in question. All fields have two modes of being accessed; either as part of the GFL structure, or as individual components.

More technical information of the implementation and usage of the GMV and GFL structures can be found in Section A.6 on page 51.

1.5 CODING STANDARDS AND CONVENTIONS

There is a comprehensive document (see Appendix F) which gives a full and complete guide to the "coding norms" to be used when writing and submitting code to the IFS system. In this section, brief highlights of some of the most important features of the coding standards are given, but it is recommended that anyone planning on writing any significant amount of code for IFS refers to the full Coding Standards document in Appendix F.

1.5.1 Style and layout

- Each file should contain only one module or procedure. The filename should be the name (in lowercase letters) of the procedure it contains, with an appropriate extension (eg. .F90 for FORTRAN 90).
- Executable lines should be written in uppercase characters, comments can use a mixture of case as appropriate (but should be in English only). A consistent style should be maintained throughout a subroutine or module.
- Use free-format FORTRAN 90, starting in column 1, but keeping lines to within 80 characters per line.
- Continuation lines are marked by the continuation character & at the end of each line to be continued *and* the start of the continuation line. Use indentation and alignment to maintain readability of long, broken lines.
- Use indentation (spaces only, no tab characters) to make the structure more obvious (ie. loops, IF blocks).
- A procedure should have only one entry and one exit point (the bottom of the procedure). Abnormal termination should be invoked with the ABOR1('Error Message') routine.
- Each data module should begin with a description of the general content of the module and the purpose of each declared variable (one line per variable).
- Each procedure should begin with comments describing:

  - the *purpose* of the procedure;
  - the *interface* details, describing the arguments in the same order they appear in the interface;
  - the *externals* (other subroutines/functions called);
  - the *method* used in the application;
  - a *reference* to further documentation;
  - the *author and date* of creation;
  - details of any *modifications* since the creation, including the author and date.
- The first and last executable statement of every subroutine should be a conditional call to DR_HOOK:
  First: `IF (LHOOK) CALL DR_HOOK('ROUTINE_NAME',0,ZHOOK_HANDLE)`
  Last: `IF (LHOOK) CALL DR_HOOK('ROUTINE_NAME',1,ZHOOK_HANDLE)`
- In a procedure, variables should be declared or USEd in the order:
  - variables USEd from modules;
  - dummy arguments (in the same order as they appear in the argument list), and using the INTENT attribute;
  - local variables.
- Loops should be written only using the DO ... ENDDO construct.
- Use the SELECT CASE construct in preference to IF/ELSEIF/ELSE/ENDIF statements.
- Use the FORTRAN 90 comparison operators rather than the FORTRAN 77 style operators (ie. "less than" should be "<" rather than ".LT.").

1.5.2 Variables

- The use of IMPLICIT NONE is mandatory.
- Each variable should be declared on a separate line, with declarations of variables with similar type and attributes being grouped together. All the attributes of a given variable should be grouped within the same instruction.
- Arrays should be declared using the DIMENSION attribute, with the shape and size of the arrays being declared inside brackets after the variable name on the declaration statement.
- The use of array syntax is not recommended, except for simple operations such as the initialisation of copying of whole arrays.
- Where a MODULE is used to import a variable into a subroutine, the ONLY attribute must be used, so that only those variables actually used by the procedure are imported.
- Derived types should be declared in a module. Such a module should contain ONLY the declaration of a single derived type (or a group of derived types if they are closely related), and any "primitive" operations on the types (such as allocation/deallocation of its components).
- All INTEGER and REAL variables and constants must be declared using explicit KIND, using the parameters defined in the modules PARKIND1 and PARKIND2. Table 1.1 shows the commonly used KIND parameters.
- The first (or first two) letters of every variable name indicate its type and scope, as described in Table 1.2. Prefixes shown in red and/or ~~crossed out~~ indicate those prefixes are not available for that particular variable type/scope.

1.5.3 Banned features

- GO TO should not be used (use instructions such as DO WHILE, EXIT, CYCLE, SELECT CASE instead).
- Use format descriptors rather than the obsolescent FORMAT statement.
- Use MODULEs rather than COMMON blocks.
- Do not change the shape or type of a variable when passing it to a subroutine.
- CHARACTER variables should be declared using the syntax `CHARACTER(LEN=`*n*`) var_name`.
- Arrays must not be declared with implicit *size* (`REAL(KIND=JPRB) :: A(*)`) but can be declared with implicit *shape* (`REAL(KIND=JPRB) :: A(:)`).

Table 1.1 *Commonly used KIND parameters.*

KIND name	SELECTED_*_KIND value(s)	Fortran 77 equivalent	Range *Approx.*	Precision
JPIS	4	INTEGER*2	$\pm 2^{15}$	–
JPIM	9	INTEGER*4	$\pm 2^{31}$	–
JPIB	12	INTEGER*8	$\pm 2^{63}$	–
JPIA[1]	9 *or* 12	INTEGER*4 *or* INTEGER*8	$\pm 2^{31}$ *or* $\pm 2^{63}$	–
JPRM	(6,37)	REAL*4	$\pm 10^{37}$	10^{-7}
JPRB	(13,300)	REAL*8	$\pm 10^{307}$	10^{-15}
JPRH[2]	(13,300) *or* (28,2400)	REAL*8 *or* REAL*16	$\pm 10^{307}$	10^{-15} 10^{-31}

[1]If 64 bit INTEGERs are available, then these are used, otherwise 32 bit INTEGERs are used.
[2]If 128 bit REALs are available, then these are used, otherwise 64 bit REALs are used.

Table 1.2 *Variable Prefix Naming Convention.*

	Fortran Type				
Scope	INTEGER	REAL	LOGICAL	CHARACTER	**Derived type**
MODULE variable	M,N	A,B,E-H, O,Q-X	L ~~LD,LL,LP~~	C ~~CD,CL,CP~~	Y ~~YD,YL,YP~~
Dummy argument	K	P ~~PP~~	LD	CD	YD
Local variable	I	Z	LL	CL	YL
Loop control	J ~~JP~~	-	-	-	-
PARAMETER	JP	PP	LP	CP	YP

1.5.4 I/O

- User supplied configuration variables should be access via a conventional formatted sequential file containing namelists (Unit NULNAM=4).
- Each namelist should be contained is a specific include (.h) file, with the filename being the same as the namelist name (in lowercase).
- Output messages should be written to unit NULOUT, error messages to unit NULERR. Do not explicitly write to units 0,6 or "*".

1.5.5 Parallelisation

- Only use MPL package for message passing, and set the CDSTRING to the name of the caller routine.