RESEARCH DEPARTMENT
MEMORANDUM

To:	Jean-Noël Thépaut	
Copy:		
From:	Deborah, Anne, John, Tomas and Mike	
Date:	July 8, 2011	File: R48.3/DS/XXXX
Subject:	**DRAFT - Preparation of IFS for OOPS**	

1 Introduction

This document describes the preparation of the IFS for OOPS. This is the so-called 'Bottom-Up' work in the OOPS development.

The OOPS-IFS will consist of a calling hierarchy with 4 distinct layers:

1. A main program written in Fortran - this will start up the Message Passing, the signal handlers, OpenMP threads, call OPENDB to open the ODB and setup some constants.
2. The OOPS object-oriented control layer written in C++ - this sits above the computational code and manipulates objects in IFS. This will replace the current top level routines CNT0-4 or CVA1-2, SIM4D, CNT[3-4]TL, CNT[3-4]AD. This C++ layer will not do any computations or allocate any memory.
3. A set of OOPS-IFS Fortran interface subroutines
4. Existing subroutines from IFS.

This document describes the preparation of the last two: the OOPS-IFS Fortran interface subroutines and the pre-OOPS preparation and cleaning of IFS. The OOPS object-oriented control layer is contained in the OOPS code released for review on 13th June 2011. The OOPS tarball also contains a prototype or toy model: a set of OOPS-QG Fortran interface subroutines and subroutines for a toy QG model written in Fortran.

Initially the OOPS-IFS work is aimed at a 3D-Var prototype.

2 Pre-OOPS Cleaning and reorganisation of IFS

Much of the pre-cleaning of the IFS code necessary to be able to isolate required data and operator objects has been included in the cleaning of IFS for Cycle 38. Various types of cleaning work have been done:

- Removal of coding norm violations: removal declared but unused variables and where possible introduction of 'USE with ONLY' for Fortran modules.

- Cleaning work as described in Karim Yessad's documents 'Proposal of cleanings in Arepege/IFS in 2011-2012' Versions V6 and V7. This includes introducing new derived types, removal of arguments to subroutines that are not used, removal of useless routines.
- For model fields replace ALLOCATABLE arrays in modules by POINTER arrays - so that there can be several instances at once when called from the OOPS layer.
- Make a start on removing the use of data from global modules and instead pass through argument lists.
- Make scalars that are constant into PARAMETERs.
- Reorganisation of IFS modules - see Karim Yessad's document 'OOPS: Action Cleaning' Version V2.

This cleaning work will continue in CY39.

3 Preparation of objects from IFS ready to plug into OOPS

Prepare Operators and data structures (and their setup) required by the OOPS object-oriented control layer.

3.1 Model Fields

3.1.1 Changes to IFS

For the modules YOMGMV and YOMGFL new derived types have been introduced mirroring the module's original contents. To minimize the changes to the IFS source, the old module variables are kept, but with any allocatable arrays changed to pointer arrays. SET and SAVE module procedures have been added which associates the old module data with with the components of the derived types (and vice versa). In this way, multiple fieldsets may be allocated, with only one at time in use by the original IFS code.

For 3D-Var, only one instance of the geometry object is requred. Hence, derived type wrapping for geometry related module data is not yet implemented.

The call to SUSC2B will have to be changed, as it also allocates arrays which belong to the fieldset object.

3.1.2 OOPS-IFS Fortran interface subroutines

The following Fortran subroutines are called by the C++ layer and call IFS Fortran routines:

```
geometry_setup
fields_setup
fields_read
```

`geometry_setup` fills the OOPS geometry object which holds all data describing grid dimensions and orography by calling the routines:

```
 SUDIM1
 SUGFL
```

```
SUAFN
SUJB
SUDIM2
SUALLO
SUGEM1A
SETUP_TRANS
SUMP
SULEGA
SULAP
SUVERT
SUGEM1B
INI_IOSTREAM
SUSC2A
SUDYN
SUSC2B
```

Currently, the OOPS-IFS fieldset object holds the contents of modules YOMSP, YOMGMV, YOMGFL and SURFACE_FIELDS_MIX.

`fields_setup` allocates a fieldset instance by calling the routines:

```
SUALSPA
SUALSPA1
SETUP_GMV
SU_SURF_FIELDS
ALLO_SURF
```

and additionally directly allocating the components of YOMGFL.

`fields_read` reads the fieldset instance after it has been allocated The reading is done by the IFS routine SUINIF, followed by a transform of upper air data from spectral to gridpoint space.

3.1.3 Setup of Constants

Non geometry dependent module data which is used by `fields_setup` and `geometry_setup` or other routines in 3D-Var, is setup in routine `ifs_init`. This routine is not part of the OOPS Fortran interface, but is called by the Fortran main program, before control is handed over to the OOPS C++ layer. It calls the following routines:

```
SUMPINI
SULUN
SUMSC
SUARG
SUCT0
ifs_constants
SUDYNA
SUMP0
SUVAR
```

```
SUFA
SUOPH
SUVWRK
SETUP_TRANS0
```

To reduce the total number of setup calls for 3D-Var, the routine `ifs_constants` called from `ifs_init` is used to set various scalar variables in the modules:

```
YOMCT3
YOEWCOU
YOMDPHY
YOMPHY
YOMCOAPHY
YOMNUD
YOMARPHY
YOMMCC
YOEPHY
YOMSIMPHL
YOMGEMS
YOPHLC
YOMOBS
YOMDYNCORE
YOMCST
YOMINI
YOMNMIA
YOMVRTL
YOMRCOEF
```

Most of those modules are related to model physics and will eventually be intialized in the yet to be written `model_setup` routine.

3.2 Interpolation from Model Grid to Observation Points

3.2.1 Changes to IFS

In IFS the observation interpolation is called from subroutines SCAN2M, SCAN2MTL and SCAN2MAD (referred to collectively by the suffix xx). Part of the preparation included placing all of the routines for observation interpolation into new routines called COBSALLxx. In SCAN2Mxx, COBSALLxx calls COBSxx, SLCOMM, SLEXTPOL, and COBSLAGxx.

A complete list of module variables required by COBSALLxx was generated by creating an executable without using the IFS libraries. A list of over 100 variables was produced.

All references to the GOM arrays in routines in the COBSALLxx calling chain were changed to be passed through calling arguments, rather than direct reference to the module. This change was also made for the various call chains referencing the GOM arrays in TASKOBxx. Approximately 100 routines were changed.

A new derived type, TYPE_GOMALL, was introduced in GOMS_MIX, which contains all references to the GOM data. YGOM and YGOM5 are instances of TYPE_GOMALL contained in GOMS_MIX.

Routine SUALOBS calls SUGOMS to allocate arrays in YGOM, and also calls SUGOMS to allocate arrays in YGOM5 if LOBSTL is set (i.e. for minimisations).

Module YOMGLOBS was modified so that the derived type TGLOBS contained all references to the message passing arrays (setup in MKGLOBSTAB). YGLOB was created as an instance of TGLOBS. A pointer to YGLOB is included in TYPE_GOMALL, so that both YGOM and YGOM5 will have access to YGLOB.

Namelist variables cannot be read directly into derived types so two variables (LGOMUA and NUSE_ECCI) were read directly into the GOMS_MIX module. LGOMUA was copied into YGOM and YGOM5 ar the end end of SUGOMS. NUSE_ECCI was made part of the derived type TYPE_GOMALL, and setting it was moved from SUDIM to SUGOM.

YGOM5 and YGOM are now arguments at the highest level of the calling chain. Wherever possible, only a single YGOM pointer is passed (rather than YGOM and YGOM5). COBSALLxx has only a single pointer. Some routines need both YGOM and YGOM5 (e.g. HOPTL and HOPAD).

COBSALLTL creates a TL version of YGOM and COBSALL creates either YGOM (for a trajectory) or YGOM5 (for a minimisation). COBSALLAD uses YGOM as input.

Module YOMECTAB (observation table definition) was split into a static part and a variable part (YOMVEC) containing the timeslot specifications.

3.2.2 OOPS-IFS Fortran interface subroutines

The following Fortran subroutines are called by the C++ layer and call IFS Fortran routines:

```
fields_interp_setup
fields_interp
fields_interp_tl
fields_interp_ad
```

In the future the Field and Geometry inputs will be provided by pointers from C++. However, currently these inputs are provided in Fortran modules. Outputs are pointers to GOM arrays which contain the interpolated values, which are passed to C++.

`fields_interp_setup` creates and initialises YGLOB as an instance of the TGLOB derived type and creates and initialises YGOM as an instance of the TYPE_GOMALL derived type. A pointer to YGLOB is put into YGOM. A pointer to YGOM is returned to C++. The routine is called once for each YGOM required. It calls IFS routines:

```
RD_OBS_BOXES
MKGLOBSTAB
SUOGBSADDR
SUGOMS
SUOBB
```

`fields_interp` calls `COBSALL` and fills GOM arrays. It receives and returns a pointer to YGOM from C++.

`fields_interp_tl` calls `COBSALLTL` and fills GOM arrays.

`fields_interp_ad` calls `COBSALLAD` and takes GOM arrays as input, returning GMV and GLF arrays as output.

3.3 Observation Vectors

3.3.1 Changes to IFS

To simplify the setup, constants in YOMCOCTP such as NSYNOP were changed to PARAMETERS. A new switch: LOOPS=.T. for OOPS and LOOPS=.F. for current IFS was introduced. For LOOPS=.T. RTSETUP was changed only to do setup related to OOPS - and return after the call to RTTVI.

3.3.2 OOPS Fortran interface layer

The following interface subroutines are called by the C++ layer and call IFS Fortran routines:

```
obsvec_setup
obsvec_read
obsvec_write
obsvec_clone
obsvec_sub
```

A new derived type `OBSVEC` is introduced in the OOPS Fortran interface layer. All observations are copied for all `KSET`s from the ODB using the HOP sql into `OBSVEC%VAL(:)` for manipulation by the OOPS layer.

`obsvec_setup` does setup necessary for observations and calls routines:

```
obsvec_constants
SET_KSET  (a simplified version of ECSET)
RTSETUP
```

`obsvec_constants` sets constants needed for observation calculations in IFS modules:

```
YOMDIMO  : NMXSET   , NMXTCH
YOMVAR   : NUPTRA   , LCLDSINK, CTOPBGE, CAMTBGE
YOMCMA   : NRESUPD
YOMOBSET : MTYPOB   ,MLNSET   ,MTSLSET, MMXBDY, NSETOT
YOMCOSJO : NPRTBINS
YOMTVRAD : SATGRP_TABLE, TOVSCVX, TOVSCVX5,  &
          & TVTSBGE_LAND,TVTSBGE_SEA,TVTSBGE_ICE, NVATOVLEN, NVALCHAN
YOMCVA   : NVATOVV
SATS_MIX : LCO2_DIAG_AIRS, LCO2_DIAG_IASI
YOMCMBDY : NCMDACBP ,NCMDACOC
YOMMKODB : NTBMAR, NTFMAR
YOMSTA   : NLEXTRAP, RDTDZ1
YOMVRTL  : LTLINT   ,LOBSTL
YOMRINC  : QNLMINQ  ,QNLCURV
```

`obsvec_read` calls GETDB/PUTDB using the HOP sql and allocates and outputs `OBSVEC` with Observations copied from the ODB.

`obsvec_clone` allocates and outputs an `OBSVEC` of the same size as the input `OBSVEC`.

`obsvec_write` calls GETDB/PUTDB using the HOP sql and saves an input `OBSVEC` back to the ODB.

`obsvec_sub` calculates the difference between 2 input `OBSVEC`s - this replaces HDEPART

3.4 Observation Equivalence operators

3.4.1 Changes to IFS

For most Observation types the Observation Equivalents are called from HOP which is called in IFS from TASKOB. For LOOPS=.T. HOP is modified to return after the calculation of ZHOFX and before the call to HDEPART (which is now done from the C++ layer by a call to `obsvec_sub`). Also for LOOPS=.T. ZHOFX is passed back as an argument to HOP. We will start by testing 3D-Var for a radiance observation type AMSU-A - the calculations of Observation Equivalents for Radiances is already well encapsulated in RTTOV. We will also start by running with LRTTOV_INTERPOL=.T. when interpolation to RTTOV levels is done inside RTTOV. This means that PPOBSA will not be called to do the vertical interpolation by calling the PP routines - this reduces the number of modules that need to be setup for HOP - but will be the subject of future work.

For MWAVE, SMOS and GBRAD Observations the Observation Equivalents are currently calculated in CALLPAR. This is being moved (initially for MWAVE) to HOP.

The GOM arrays have been modified to be passed through the argument lists in the HOP tree as described above.

3.4.2 OOPS Fortran interface layer

The following Fortran interface subroutines are called by the C++ layer, and call IFS Fortran routines.

```
obs_equiv
obs_equivtl
obs_equivad
```

`obs_equiv` calls `HOP` in a LOOP over `KSET`s. It has input a pointer to the GOM array and outputs an `obsvec` containing Observation Equivalents - this replaces TASKOB

`obs_equiv_tl` calls `HOPTL` in a LOOP over `KSET`s. It has input a pointer to the GOM array and outputs an `obsvec` containing Observation Equivalents - this replaces TASKOBTL.

`obs_equiv_ad` calls `HOPAD` in a LOOP over `KSET`s - this replaces TASKOBAD.

3.5 Testing of OOPS-IFS components

A simple C++ program called mini-OOPS was written to test the components of OOPS-IFS as they were written using ICM initial files and an ODB containing AMSU-A data taken from a first trajectory run of IFS T42 4D-Var. So far the mini-OOPS test C++ has tested the calculation of departures using the following sequence.

```
geometry_setup
fields_setup
fields_read

fields_interp_setup
fields_interp

obsvec_setup
obsvec_read
obsvec_clone
obsvec_sub
```

3.6 Encapsuation of JB Operator in IFS

3.6.1 IFS Modules used by J_b

J_b is reasonably self-contained. It involves around 170 subroutines, with dependencies on around 30 modules. The subroutines involved are listed in table 1. The modules and the variables USEd from each module are listed in table 2

The modules can be divided into three groups. The first group contains modules that can be regarded as constant throughout an execution. This group is listed in table 3. These modules can be left as global variables since they will be the same for all instances of J_b.

The second group of modules can be regarded as belonging purely to J_b. These modules are listed in table 4. Modules in this group may change from one instance of J_b to the next. For this reason, the global variables in these modules must be gathered into derived types, so that each instance of J_b will have its own variables.

The third group of modules contain variables that vary between instances of J_b, but are not "owned" by J_b. These are listed in table 5. These modules correspond to the "geometry" and "model" objects. Each instance of J_b will contain a pointer to the Fortran representation of one of these objects (i.e. to an instance of a derived type).

A number of messy areas remain in this categorisation. Some modules contain variables that belong to more than one class. These modules will have to be split so that the ownership of each variable is clear.

3.6.2 Reorganisation of J_b modules into derived types

As discussed above, the modules owned by J_b must be moved into derived types. This is to allow instances of J_b to be different (e.g. at a different resolution).

The proposed structure is to divide the module contents of `YOMJG`, `YOMWAVELET`, etc. into more than one derived type. For each of these modules, one type will contain variables that can be specified in the namelist.

This is necessary because Fortran does not allow elements of derived types to be specified in a namelist. Hence, for each module, the namelist read must specify the name of an entire derived type. By splitting the current namelist variables off into their own derived type, we restrict the variables that can be set via the namelist to those in the current namelists.

A second derived type in each module will contain data that is not accessible via the namelist. Other derived types correspond to those already in existence.

The collection of derived types will be further organised into a single over-arching derived type for J_b, which will be defined in `YOMJG`, and will look something like this:

```
TYPE TYPE_JB_STRUCT
  TYPE (TYPE_JB_CONFIG)                            :: CONFIG
  TYPE (TYPE_JB_CONFIG2)                           :: CONFIG2
  TYPE (TYPE_JB_DATA)                              :: JB_DATA
  TYPE (TYPE_FCEMN_STRUCT)                         :: FCEMN
  TYPE (TYPE_VCORS_STRUCT),          POINTER       :: VCORS(:)
  TYPE (TYPE_SPJB_VARS_INFO_STRUCT), POINTER       :: SPJB_VARS_INFO(:)
  TYPE (TYPE_WAVELETJB_CONFIG)                     :: WJBCONF
  TYPE (TYPE_WAVELETJB_DATA)                       :: WJBDATA
  TYPE (TYPE_WAVELETJB_VCOR_STRUCT),   POINTER :: WAVELET_VCORS(:,:)
  TYPE (TYPE_WAVELETJB_GRID_STRUCT),   POINTER :: GRID_DEFINITION(:)
  ...
END TYPE TYPE_JB_STRUCT
```

Here, each element corresponds to one of the derived types defined in `YOMJG`, `YOMWAVELET`, etc. `CONFIG` and `CONFIG2` correspond to the namelists `NAMJG` and `NAMJBCODES`. (Note that `NAMJBCODES` must be read after `NAMJG`, so these namelists cannot be combined.) A new namelist, `NAMWAVELETJB` will be added, corresponding to the "wavelet-J_b" variables in `WJBDATA`.

Once the "geometry" and "model" objects are defined, additional pointers will be included in `TYPE_JB_-STRUCT`, pointing to derived types for the geometry and model.

3.6.3 Removal of Global Variables

There are two approaches that can be taken for the global variables for which a corresponding variable exists in, or below, `TYPE_JB_STRUCT`. One approach, which leads to the least modification of existing subroutines, is to continue to define the global variables, and to use them in the code. With this approach, which I will call "Tomas's trick", the variables in the derived type must be copied into the global variables on entry to any of the J_b methods (e.g. before a call to `CHAVARIN`). (This extra copying is not too large an overhead provided global arrays are converted to global pointers, so that copying the contents of an array can be replaced by reassigning a pointer.)

However, Tomas's trick is somewhat dangerous. For example, it is easy to forget to copy a global variable. If this happens, it is possible for one instance of J_b to modify another instance. Unless some form of automatic checking is implemented, Tomas's trick has the potential to lead to some very difficult-to-resolve bugs.

For this reason, I recommend that all global variables included under `TYPE_JB_STRUCT` should be removed. A consequence is that all references to variables currently in `YOMJG`, `YOMWAVELET`, etc. must be converted to references to elements of derived types.

To ease the transition from the current J_b, I suggest a single global pointer `JB_STRUCT` be defined, corresponding to an instance of `TYPE_JB_STRUCT`. This global pointer will be set on entry to any of the J_b methods. (I.e. we use Tomas's trick, but limit it to this single global variable.)

Initially, using `JB_STRUCT` will pepper the code with some rather long variable names (for example, `LJBWAVELET` becomes `JB_STRUCT%WJBCONF%LJBWAVELET`). Ultimately, it is expected that this will encourage people to pass individual variables (e.g. `LJBWAVELET`), or sub-types of `JB_STRUCT` (e.g. `WJBCONF`) through the argument list. It should be a long-term aim to remove the need for the global variable `JB_STRUCT` altogether.

3.6.4 Consequences for Namelists

As discussed above, Fortran does not allow elements of a derived type to be read via a namelist. This leaves two options. The first is to leave the namelist as it currently is. In this case, reading the namelist becomes a two-stage process. Each namelist variable corresponds to a local variable in some namelist-reading subroutine. This subroutine would copy the derived-type components to the local variables before the namelist read. Reading the namelist would modify some of these local variables, which would then be copied back into the derived type.

This option has the advantage that the current namelists could be retained. However, it requires careful maintenance of the correspondence between local variables and their derived type counterparts.

The preferred option, is to reduce the namelist to the name of the corresponding derived type (`CONFIG`, `CONFIG2`, `WJBCONF`, etc.). The namelist file would have to be suitably modified to specify the components (e.g. `REDNMC` becomes `CONFIG%REDNMC`). Each derived type requires its own namelist, so that variables for wavelet J_b could no longer be specified via `NAMJG`, but would require their own namelist.

A consequence of this option is that the namelists become trivial. Each consists of a single variable name corresponding to the name of a variable of a derived type. In view of this, it is recommended that the namelist include files for these variables (e.g. `namjg.h`) should be removed. A small subroutine to read each of the J_b namelists will be added. these subroutines could be `CONTAIN`ed routines either in corresponding module, or in `sujb.F90`.

3.6.5 Areas of Entanglement

1. Dependence on the model geometry.
2. Dependence on the model state (trajectory and background).
3. Trajectory and background not always clearly identified, with some use of model arrays for temporary storage.
4. The energy-norm J_b (`LJBENER=T`) uses variables from the dynamics and is partially set up in `SUDYN`.
5. `SPCHOR` (stochastic physics) uses the J_b balance operators.
6. The multiplication by $B^{1/2}$ and the addition of the background should be separated. `LSUBBG` appears in far too many rouines.
7. `READVEC` works out whether to read ozone fields based on flags `LFCOBS` and `LSKF`. Should pass in a flag.

balnonlin	balnonlinad	balnonlintl	balomega
balomegaad	balomegatl	balstat	balstatad
balvert	balvertad	balverti	commfce2
commjbbal	commjbdat	copy_spa2spec	copy_spec2spa
cvar3in	cvar3inad	cvargpad	cvargptl
ebalbeta	ebalbetaad	ebalnonlinad	ebalnonlintl
ebalomegaad	ebalomegatl	ebalstat	ebalstatad
ebalvert	ebalvertad	eigenmd	einv_trans
ejbwav_cv2wav	ejbwav_gp2wav	ejbwav_h2v	ejbwav_hcori
ejbwav_v2h	ejbwav_vcori	ejbwav_wav2cv	ejbwav_wav2gp
ejghcori	ejgnrgg	ejgnrggad	ejgnrggi
ejgnrggiad	esperee	etransdir_jb	etransdir_jbad
etransinv_jb	etransinv_jbad	fltbgcalc	gatherbdy
gathereigmd	gatherspa	inifger	jbchavari
jbchvariad	jbmatinterp	jbvcoord_interpolate	jbvcoord_interpolate_ad
jbvcor_waveletin	jbvcor_waveletinad	jbvcorg	jgcori
jgcoriad	jghcori	jghcosi	jghcosiad
jgnr	jgnrad	jgnri	jgnriad
jgnrsi	jgvcor	objtrunc	pe2set
pregprh	qasset	qneglim	read_spec
readvec	sc2rdg	sc2wrg	scalefe
set2pe	spec_imzero	speree	sqrtbin
sqrtbinad	sqrtfe	sqrtqin	sualges
sualspajb	suejbbal	suejbcov	suejbwavelet
suejbwavelet_bmatrix	suelges	suhifce	suinfce
sujb	sujbbal	sujbchvar	sujbcor
sujbcosu	sujbcov	sujbcovnoise	sujbcovsignal
sujbdat	sujbgptomat	sujbstd	sujbwavallo
sujbwavalls	sujbwavelet	sujbwavelet0	sujbwavgen
sujbwavstats	sujbwavtrans	sujbwavvc	sujbwavwri
sumdfce	suprecov	suprffce	susepfce
sushfce	suskf	suvifce	transdir_wavelet
transdir_waveletad	transinv_wavelet	transinv_waveletad	troplev
vec2gpfe	wavxform	write_spec	

Table 1: List of subroutines involved in J_b. Some low-level routines have been omitted.

CONTROL_VECTORS	CONTROL_VECTOR
GRIDPOINT_BUFFERS_MIX	GRIDPOINT_BUFFER, ALLOCATE_GRIDPOINT_BUFFER, DEALLOCATE_GRIDPOINT_BUFFER, ALLOCATED_GRIDPOINT_BUFFER
GRIDPOINT_FIELDS_MIX	GRIDPOINT_FIELD
SPECTRAL_FIELDS	SPECTRAL_FIELD, ASSIGNMENT(=), ALLOCATE_SPEC, DEALLOCATE_SPEC
SURFACE_FIELDS_MIX	TYPE_SURF_MTL_2D, TYPE_SFL_VARSF, SD_VF, YSD_VF
YOM_YGFL	JPGFL, JPGHG, JPGRG, NAERO, NGRG, NGHG, YGFL, YGFLC, YI, YL, YO3, YQ, YAERO, YGRG, YGHG
YOMCOSJB	FJBCOST, LJBZERO, LJPZERO, LJLZERO, LJTZERO
YOMCT0	LMPOFF, NPROC, N_REGIONS_NS, N_REGIONS_EW, NPRGPNS, NPRGPNS, N_REGIONS, NPRTRW, NCONF, LELAM, LNHDYN, NPRTRV, LBACKG, LOBSC1, LGUESS, NCNTVAR, LSPBSBAL, LSIMOB, LCANARI
YOMCT0B	LECMWF
YOMCT3	NSTEP
YOMCVA	NVA3SP, NVA3D, NVA2D, NVA1D, NGRBVAR, YRSCALP
YOMDIM	NDGLG, NDGNH, NPROMA, NFLEVL, NFLEVLMX, NFLEVG, NFLSUR, NSMAX, NMSMAX, NSPEC2G, NSPEC2, NSPEC2MX, NUMP, NUNDEFLD, NGPBLKS, NFTHER, NS3D, NS2D, NTPEC2, NCPEC2, NCMAX, NTMAX, NPPM
YOMDYN	SITR, SIDELP, SIRPRG, SIRPRN
YOMFCEB	FCEBUF, GPBUF_JBVCOORD, BLTOP
YOMFGER	JPMAXLAT, JPMAXLEV, JPMAXVAR, MNROWS, MNVARS, MNFLDS, MNPTE, MNLEVMX, MLONE, MFIELDS, MNLEVS, RZLATE, RZLON0E, RZDLONE, RZA, RZB, RZEGRID
YOMGC	RCORI, GEMU, GSQM2, GELAM, GELAT, GM, GAW
YOMGEM	NGPTOT, NGPTOTG, NLOENG, NSTAGP, VP00, VAH, VBH
YOMGMV	TYPE_T0, YT0, YT5
YOMGRB	NGRBU, NGRBBLH, NGRBGHG, NGRBGRG
YOMJBCHVAR	RQS, RQT, RQTBAL, RQLMIN, RQLMAX, RQLSTD, RQHMIN, RQHMAX, RQHSTD, RO3MIN, RO3STD
YOMJG	Too many variables to include in this table.
YOMLAP	NASM0, NASM0G, RLAPDI, RLAPIN, RCAPDI, MYMS, NVALUE, NSTREL, NSPINDXG
YOMLCZ	COEQTERM
YOMLEG	RW, RMU
YOMLUN	NULNAM, NULOUT, NULERR, NULTMP, NULDILA, NULCONT
YOMMP	MYPROC, NPSP, NSPEC2V, MYSETW, MYSETN, MYSETV, MY_REGION_NS, MY_REGION_EW, NBSETSP, NPRCIDS, NUMPP, NPROCM, NALLMS, NPTRMS, NUMLL, NPTRLL, MYLEVS, NPSURF, NSTA, NONL, NPTRFRSTLAT, NFRSTLAT, NLSTLAT, NBSETLEV, NPTRSV
YOMMPI	MINTET, MREALT, MLOGIT, MCHART
YOMMSC	Z_DEFAULT_REAL, N_DEFAULT_REAL_KIND, DL_DOUBLE_PRECISION, N_DOUBLE_KIND
YOMNMIA	NLEX, NITNMI, NVMODF, NVMODF1, NVMODF2, NVMODM, NVMOD, NVMODJB, LNMIRQ, LRPIMP, LCONMO, LCOIMP
YOMPLDSW	LOPT_SCALAR, LOPT_RS6K
YOMRINC	SPA3I, SPA2I, QNLMINQ, QNLCURV
YOMSCC	LSCDPR
YOMSKF	LSKF
YOMSPJB	SPJB, GP7A3, SP7VOR, SP7T, SP7Q, SP7SP
YOMSTA	STPRE, STTEM
YOMTAG	MTAGFCE, MTAGCAIN, MTAGEIGMD
YOMVAR	LINITCV, LFCOBS
YOMWAVELET	LJBWAVELET, LJBWSTATS, L_IN_WAVELET_STATS_CALC, N_WAVELET_SCALES, N_BGMEMBERS, N_BG-DATES, NMAX_RESOL, NSMIN_WAVELET, SKYLINE_TOL, N_WAVELET_CUTOFFS, WAVELET_FILTERS, CINBGSTATES, WAVELET_VCOR_STRUCT, WAVELET_VCORS, WAVELET_GRID_STRUCT, GRID_DEFINI-TION

Table 2: List of modules USEd by J_b together with the variables USEd. Note that YOMHOOK, YOMGSTATS, YOMCST and YOM_GRIB_CODES are also USEd, but have been excluded from the list.

YOMLUN
YOMHOOK
YOMCST
YOMGSTATS
YOM_GRIB_CODES
YOMTAG

Table 3: List of modules USEd by J_b that are constant for all instances of J_b.

YOMJG
YOMWAVELET
YOMCOSJB
YOMSKF
YOMSPJB
YOMJBCHVAR
YOMFGER
CONTROL_VECTORS
YOMCVA

Table 4: List of modules USEd by J_b that belong purely to J_b.

YOM_YGFL
YOMDIM
YOMDYN
YOMGEM
YOMGMV
YOMLAP
YOMMP
YOMNMIA
SURFACE_FIELDS_MIX

Table 5: List of modules USEd by J_b that belong to other objects.