Frequently Asked Questions

What are namespaces?

A namespace in ecCodes is just a fancy word for a grouping of related keys. The following are available:

- `ls:` This is the namespace used by the grib_ls and bufr_ls tools and has the most-commonly used keys e.g. centre, shortName, level etc
- `parameter:` Contains keys like paramId, shortName, units which relate to the meteorological parameter
- `statistics`: Contains keys that relate to the statistics of the data values e.g. maximum, minimum, average, standard deviation etc
- `time`: Contains keys that describe the forecast runs e.g. forecast date, validity date, steps etc
- `geography`: Contains keys that describe the grid geometry e.g. bounding box of the grid, number of points along a parallel etc
- `vertical`: Contains keys that describe the levels and layers e.g. type of the level, list of coefficients of the vertical coordinate etc
- `mars`: Contains the list of MARS (ECMWF's Meteorological Archive and Retrieval System) keywords like class, stream, type etc

The contents of a namespace can vary depending on the type of GRIB message, for example the `geography` namespace will contain the `pl` key for reduced grids but not for regular grids.

Why am I not able to set the missing value in the GRIB message?

Missing values cannot be encoded in a GRIB message. The GRIB format does keep track of missing values but through the use of a **bitmap** It does not allow the specification of a missing value. Setting the missing value is a feature that can be used only when encoding the data values stored in a GRIB message. This of course means that is the responsibility of the user to know what missing value is meaningful to the data. A default value of 9999 is set for the missing value in the library (not the GRIB message!). That means that when retrieving the values

from a message without having set the missing value key, all missing values in the data will be replaced with the default value of 9999.
A small example on the use of the missing value during encoding can be found here: grib_set_bitmap.
During decoding it is advisable to query the bitmap directly to discover missing data values. See example here: grib_iterator_bitmap

How do I detect keys whose values are missing in a BUFR message?

Each element in the data section of a BUFR can be missing. ecCodes provides a simple way for the user to check if the value of an element is missing by comparing with two constants:

```
CODES_MISSING_LONG
CODES_MISSING_DOUBLE
```

These constants are available in Python, Fortran 90 and C and the user needs to compare with the appropriate constant depending on the type of the variable used. This is an example of Fortran 90 code:

```
!declare backscatter as double precision
real(kind=8), dimension(:), allocatable :: backscatter

!get the values
call codes_get(ibufr,'/beamIdentifier=2/backscatter',backscatter);
do i=1,size(backscatter)
    !compare with double precision missing
    if (backscatter(i)/=CODES_MISSING_DOUBLE) then
        !process non missing values
    else
        !process missing values
    endif
end do
```

BUFR: What is a "subset"?

A "data subset" is defined as the subset of data described by one single application of the collection of descriptors. In this context, the "collection of descriptors" means ALL the descriptors included in section 3 of the BUFR message.
In other words, one pass through the complete collection of descriptors will allow one to decode one data subset from section 4. One then loops back in the descriptor list for as many times as indicated by the number of data subsets (key "numberOfSubsets"). All the data in section 4 are properly described by repeated use of the same set of descriptors from section 3.

How do I append messages to an existing GRIB file?

Try using grib_open_file (Fortran interface) with mode "a" instead of "w". The "a" mode means: *Append to a file. Writing operations append data at the end of the file. The file is created if it does not exist*.

How do I delete a message from a file?

The tools never change the input file(s). You have to create a new one without the offending message and then do a rename. E.g.

```
# Delete the fifth message in the file orig.grib (count = 5)
% grib_copy -w "count!=5" orig.grib temp
% mv temp orig.grib
```

This can also be done using the grib_filter (or bufr_filter) where a more complex condition can be entered.

How do I copy a selected number of messages from a GRIB file?

Say you want to copy the first 3 messages. This can be done with:

```
% grib_copy -w count=1/2/3 in.grib out.grib
```

But it is better to write a rules file and use grib_filter:

my.filter

```
if (count < 4) {
  print "Copying message number [count]";
  write;
}
```

Then run this as follows:

```
% grib_filter -o out.grib my.filter in.grib
```

Of course you can make the IF condition more complicated e.g. select a range of messages to be copied:

```
if (count == 3 || count == 13) {
  write;  # Write out message 3 and 13 only
}
```

What's the difference between 'count' and 'countTotal'?

The "count" key is the message number in a given file whereas "countTotal" is the message number in a set of files. The former gets reset for every input file on the command line but the latter keeps on increasing.
Here is an example:

```
% grib_count -v file1 file2
      1 file1
      5 file2
      6 total
% grib_ls -p count,countTotal,shortName file1 file2
file1
count       countTotal  shortName
1           1           t
1 of 1 messages in file1

file2
1           2           msl
2           3           tcc
3           4           tcw
4           5           str
5           6           2t
5 of 5 messages in file2

6 of 6 total messages in 2 files
```

So the count for the 1st message of the 2nd file is 1 (reset) but countTotal is 2.

How can I change from second_order to simple packing?

You may be tempted to try it like this:

```
% grib_set -s packingType=grid_simple second_order.grib simple.grib
```

But this will not work and issue errors re codedValues and bitsPerValue! You need to use the repacking option as shown:

```
% grib_set -r -s packingType=grid_simple second_order.grib simple.grib
```

Also see the code example: grib_precision

What is an Octahedral Reduced Gaussian Grid?

See here.

Issues converting from grid_complex_spatial_differencing to simple packing?

There are GRIB messages whose packingType is "grid_complex_spatial_differencing" where the missing values *are not in a bitmap* but stored in the Data Section (as coded values).
In these cases conversion to simple packing can fail.

A workaround is to set the key "bitmapPresent" to 1 before the conversion. This ensures a bitmap is created for the output GRIB and the missing values correctly stored:

```
% grib_set -r -s bitmapPresent=1,packingType=grid_simple in.grib out.
grib
```

After the conversion, check the statistics of the input and output files to ensure correctness:

```
% grib_ls -n statistics in.grib out.grib
```

This will still work even if the input did not have any missing values.

How do I set the product definition template in a GRIB2 message?

Let's say you want your message to use the product definition template 40 (The full list is defined in Code table 4.0):

```
% grib_set -s productDefinitionTemplateNumber=40 input.grib2 output.
grib2
```

Now you can do a "grib_dump -O" to inspect the section 4 (Product definition section) of the output.grib2 file to see the new key "constituentType" (For other templates you will see other keys added).

What happens when I set the packingType to "grid_jpeg" on a GRIB1 message?

Actually nothing! The JPEG packing is not supported for GRIB edition 1 but rather than fail, the library leaves the message as is. This was designed for cases where a file (or set of files) containing a mix of grib1 and grib2 messages is being processed to use JPEG encoding. We convert grib2 messages and leave the grib1 messages alone.

How do I know what the native type of a key value is?

For the moment there is no simple way of knowing what the type of a key value is. However, the library provides a function that can be used to find out what the native type of a key value is. The function, available only in the C API, is called 'grib_get_native_type'. Possible key value types are the following: undefined, long, double, string, bytes, section, label or missing. It is not possible at the moment to know if the value of a key is of type scalar or vector.

How do I use an OR condition (logical disjunction) in a "where" clause?

For example in grib_get you may want to show all messages which have level of 100, 150 or 200. The way to do this is to use the "/" (forward slash) character in the "where" clause (for the -w switch)

```
% grib_get -w level=100/150/200 ...
```

This can be combined with an "AND" condition (logical conjunction). So if I want to show all messages whose level is either 0 or 1 and whose type is "an":

```
% grib_ls -w level=0/1,dataType=an ...
```

This would be equivalent to the pseudo-code:

```
if (dataType == "an" AND (level == 0 OR level == 1)) ...
```

grib_filter/bufr_filter: How can I check if a key exists (is defined) in a message?

Easy peasy. Use the "defined" function which takes a single key name as argument. e.g.

grib_filter example

```
if (defined(Latin1)) {
    print "Key Latin1 is there for this message";
}
```

grib_filter/bufr_filter: How do I control the format of array printing?

One can add several 'modifiers' to a key in square brackets to control number of columns, separator string and format:

```
[key!c%F'S']
 c -> number of columns
 F -> C-style formatting
 S -> separator string
```

Examples:

```
[latitude!11] -> print array using 11 columns
[latitude','] -> print array with entries separated by a comma
[latitude%.5f] -> print real numbers with 5 decimal places
```

Note: to print all the values on one line, set the number of columns to 0 (zero).

grib_filter/bufr_filter: Do I always need to write a rules file?

No. You can read the filter rules directly from the standard input (stdin) by using the "-" (a single hyphen) instead of the rules file path. For example:

```
% echo 'set edition=2; print "[centre]"; write;' | grib_filter -o out.
grib - in.grib
```

Or like this:

```
% grib_filter -o out.grib - in.grib <<EOF
set edition=2;
print "[centre]";
write;
EOF
```

How do I know if I am dealing with a multi-field GRIB file?

At the moment, the only possible way of detecting if a GRIB file is multi-field is by using the grib tools or the API to turn the support for multiple fields in a single GRIB message on or off and observe the results. For example you can do a "grib_count" and compare the result with "grib_ls", in a multi-field GRIB file the latter will report more messages than the former.

For example:

```
% grib_count my.grib2
30
% grib_ls my.grib2
edition  centre  date         dataType  gridType    stepRange
typeOfLevel    level  shortName  packingType
2        kwbc    20061128     fc        regular_ll  6
isobaricInhPa  100    clwmr      grid_jpeg
2        kwbc    20061128     fc        regular_ll  6
isobaricInhPa  500    5wavh      grid_jpeg
2        kwbc    20061128     fc        regular_ll  6
isobaricInhPa  1000   u          grid_jpeg
2        kwbc    20061128     fc        regular_ll  6
isobaricInhPa  1000   v          grid_jpeg
...
56 of 56 messages in my.grib2
```

Here we have a file with 30 messages (result of grib_count) but grib_ls reports 56 because by default the grib tools 'expand' the messages i. e. convert the multi-field messages to single-field ones before printing. This feature can be turned off by specifying the "-M" option:

```
% grib_ls -M my.grib2
edition  centre  date         dataType  gridType     stepRange
typeOfLevel    level  shortName  packingType
2        kwbc    20061128     fc        regular_ll   6
isobaricInhPa  100    clwmr      grid_jpeg
2        kwbc    20061128     fc        regular_ll   6
isobaricInhPa  500    5wavh      grid_jpeg
2        kwbc    20061128     fc        regular_ll   6
isobaricInhPa  1000   u          grid_jpeg
2        kwbc    20061128     fc        regular_ll   6
isobaricInhPa  975    u          grid_jpeg
...
30 of 30 messages in my.grib2
```

The use of the multi-field feature is not recommended.

Why can I read certain parameters using the grib tools, but not from my program?

Your file may be encoded as a multi-field grib file. By default, grib tools have multi-field support enabled and the API disabled. Check the examples to know how to enable this feature on your program:

- For Fortran77: multi_fortran.F
- For Fortran90: multi.f90
- For C: multi.c

Confused about stepUnits?

The key stepUnits is a transient key and it is not written in the grib message. So, you cannot expect to get the same stepUnits as when you have encoded the data. By convention, when encoding, ecCodes will choose the best unit to encode the step in so that it fits the space available in the grib message. When decoding, ecCodes will return the step in hours by default.

Note: The "step" key (as well as startStep, endStep and stepRange) were added purely for the MARS system here at ECMWF and MARS only deals with steps in units of hours.

The stepUnits can only be set in the following situations:

- before encoding, in order to tell ecCodes what units are we dealing with
- when decoding, in order to tell ecCodes what units to get the step in

Examples:

```
grib_ls and grib_get will return the step in hours by default
(stepUnits=h)

   % grib_set -s step=3600 file.grib out.grib
   % grib_get -p startStep,endStep,stepRange,stepUnits:s,stepType out.
grib
   3600 3600 3600 h instant
   % grib_set -s stepUnits=m,step=3600 file.grib out.grib
   % grib_get -p startStep,endStep,stepRange,stepUnits:s,stepType out.
grib
   60 60 60 h instant

To get the step in the units we want, one could use grib_filter like
this:

   % cat step.filter
   set stepUnits="m";
   print "[startStep] [endStep] [stepRange] [stepUnits:s] [stepType]";

   % grib_filter step.filter out.grib
   3600 3600 3600 m instant
```

One can also set values in a grib message before printing its contents with tools like grib_ls and grib_get.

```
  % grib_ls -s stepUnits="m" -p startStep,endStep,stepRange,stepUnits:s,
stepType out.grib
```

Confused about key types?

Some keys like shortName are strings (native type string) and it doesn't make any sense to set them as an integer or float as they are not possible values for the key.

Some keys are native type integer like perturbationNumber and you can set/get them as integer, float or string as you like.

Some keys are codetables and for them you have a code (integer) and an abbreviation (string) like for example "centre". You can set them as a string or as an integer, not as a float because there is no mapping for those keys into a float. An example is:

```
% grib_set -s centre:s=ecmf  in.grib out.grib  # Set the key 'centre'
as a string
% grib_set -s centre:i=98    in.grib out.grib  # Set the key 'centre'
as an integer
```

Some keys are *concepts* and they are usually strings like stepType, gridType, packingType. Again setting them as integer doesn't make any sense as they don't have any mapping as integers.

A concept is a key with a special association with other keys. Let's take shortName as an example. Its definition in the .def files is made with a long list of entries like:

```
'cape' = {
     discipline = 0 ;
     parameterCategory = 7 ;
     parameterNumber = 6 ;
     typeOfFirstFixedSurface = 1 ;
     typeOfSecondFixedSurface = 8 ;
}
```

this means that if you set shortName="cape" (as a string) ecCodes will automatically set the keys listed in curly brackets to the corresponding values.

Conversely when you get shortName, ecCodes looks for the best match of the keys listed in the full definition of shortName (not only cape) and will return "cape" only if the following are true:

```
discipline == 0
parameterCategory == 7
parameterNumber == 6
typeOfFirstFixedSurface == 1
typeOfSecondFixedSurface == 8
```

in the message and there isn't a better match of keys in the list.

Again saying shortName=123 doesn't make any sense as in the context of shortName we don't need numbers.

The idea is that the user should know what to set before choosing the type to use. In the sense that the user should know the meaning of the key before setting it to a value. For this purpose we have built the documentation and I agree that we need to do a big review (with your help) of it to fix all the wrong things and to add what is missing.

There was a bit of confusion regarding the type of step and paramId for the following reason:

step

In the past step was a string because you can have values like 24-36 which you cannot represent as numbers. In MARS the step is only the endStep and therefore we had to modify ecCodes to be MARS compliant. Now the step is an integer and is step=endStep. As we need a step indicating a range we introduced stepRange which is a string and cannot be set as an integer and will never be set as an integer because it allows values like stepRange=24-36. It is true that most of the time you have stepRange=36 or stepRange=24 or another single number, but this is because in those cases you have also stepType=instant and endStep=startStep and we don't want to write

stepRange=24-24 which is redundant. We also write stepRange=24 when stepRange=0-24 (MARS compatibility). If you prefer to set the step as an integer instead of using a string you have starStep, endStep which are integers and they can also be set as string because it is possible to convert for example the number 24 into the string "24". Therefore you can do:

```
% grib_set -s endStep=24 in out
% grib_set -s endStep:s=24 in out
```

paramId

It is a concept like shortName and unfortunately in the previous version the native type for a concept was string even if the meaning of it is number. In the new version paramId is still a concept, but it has native type number. This means that it can be set as a number, but cannot be set as a string because only the numbers are valid values for paramId. You can still get it as a string because it is possible to convert a number into a string. You cannot set it as a string because it isn't always possible to convert a string into a number.

Why I cannot set negative values for the longitude in GRIB 2

GRIB 1 regulates that the longitude can be in either [-180, 180] or [0,360], GRIB 2 regulates that the longitude can only be in the interval [0,360]. ecCodes does comply with these regulations and in the case of GRIB 2, it will scale the longitude to fall in the interval [0,360]. ecCodes does not offer an edition independent view of the longitude because of the uncertainty brought by GRIB 1, where you do not know what interval a longitude is in exactly.

For GRIB edition 1, why does the centre change when I set stepType?

This happens for stepType of "max" and "min". Unfortunately edition 1 does not support maximum and minimum in its "Time Range Indicator" table (Table 5). So we *invented* our own centre-specific combination to support this. There is no such issue in GRIB edition 2.

Why is the "bottomLevel" of Soil Temperature level 4 reported as MISSING?

The parameter 236 (Soil temperature level 4) specifies that its top level is 100cm and its bottom level is 289cm so to encode this information in GRIB edition 1 we would need to set the value of the key "bottomLevel" to 289 (octet 12 in section 1. See GRIB edition 1 section 1 spec). However since this key is only *one octet* it cannot accommodate any value larger than 255! Therefore it is not possible to encode levels larger than that value in GRIB1 so we simply set all the bits to 1 (which means MISSING).

GRIB edition 2 addresses this shortcoming.

Possible GOTCHA with wave spectra fields

In the wave spectra, all frequency/direction components with values less than some (field-dependent) threshold are set to 'missing' when encoded in GRIB; values larger than this threshold are encoded as log10 of the actual value to an accuracy of 9 bits (bitsPerValue=9).

To interpret these values correctly, the user should do:

```
if (value == missingValue) then
    value = 0.0
else
    value = 10.0**value  ! 10 to the power of the value
endif
```

What environment variables are there?

Here are the environment variables that are used by ecCodes:

ECCODES_DEFINITION_PATH: Set to the list of directories containing the set of definition files you want to use instead of the default one.

ECCODES_SAMPLES_PATH: Set to the list of directories containing the set of sample files you want to use instead of the default one.

ECCODES_DEBUG: If set to -1, will enable brief debug-level logging messages to be displayed by the library. If set to 1, you will get very verbose output.

ECCODES_FAIL_IF_LOG_MESSAGE: If set to 1, will cause the library to exit when an error or warning is encountered.

ECCODES_IO_BUFFER_SIZE: Defines the size in bytes of the buffer used in the IO calls from Fortran and in the tools.

ECCODES_NO_ABORT: When set to 1 it causes ecCodes not to abort execution on failing asserts.

ECCODES_GRIB_WRITE_ON_FAIL: When set to 1 it will write the last processed GRIB message to a file named $PID_$FILEID_error.grib on failure in a fortran function used without the return code argument.

ECCODES_GRIBEX_MODE_ON: When set to 1 it will enable the GRIBEX compatibility mode and ecCodes will produce GRIB messages readable by GRIBEX.

ECCODES_GRIB_IEEE_PACKING: Accepted values 32 or 64 for 32 or 64 bits IEEE floating point respectively. The GRIB message produced will contain data written in IEEE floating point without packing.

ECCODES_LOG_STREAM: This controls the output stream for warning/error messages. Accepted values are "stdout" or "stderr" (default is stderr)

ECCODES_BUFRDC_MODE_ON: This is for BUFR decoding functionality. When set to 1 it will enable the BUFRDC compatibility mode (e. g. we tolerate problems like wrong data section length)

ECCODES_BUFR_SET_TO_MISSING_IF_OUT_OF_RANGE:This is for BUFR encoding functionality. When set to 1 ecCodes will allow out-of-range values (BUFRDC compatibility mode). In this mode encoding does not fail and a missing value is encoded in place of the out-of-range value.

ECCODES_PYTHON_NO_TYPE_CHECKS: This is for the Python interface. When set to 1, the types of the arguments passed to ecCodes Python functions are not checked. This can improve performance specially in cases where there are a lot of calls to these functions.

If the library is built with only **one** of the jpeg libraries (jasper or openjpeg) it will work without any environment variable. If ecCodes is built with **both**, then you have always to link both and you can switch from one to the other with the variable:

ECCODES_GRIB_JPEG=jasper
ECCODES_GRIB_JPEG=openjpeg

This env. variable was only for debugging/development purposes, but it can be used by users. There is also another variable ECCODES_GRIB_DUMP_JPG_FILE=filename which provides a dump of the JPEG image on a separate file, again for debugging purposes.

How can I remove the PV array (list of vertical coordinates)?

This can be done by setting the key "deletePV" as shown:

```
% grib_set -s deletePV=1 in.grib out.grib
```

Which is in fact equivalent to setting the number of coordinate values to 0 and clearing the "pv" array:

```
% grib_set -s PVPresent=0,NV=0 in.grib out.grib
```

Failure setting key changeDecimalPrecision

If you issue the command "grib_set -s changeDecimalPrecision=1 spectral.grib output", you can get an error

```
ECCODES ERROR   :  COMPLEX_PACKING : Cannot compute binary_scale_factor
```

This is because that key does not work for **spectral_complex!** So first check the packingType key.

grib_compare: Is there an alternative to blacklisting keys?

For example if you are interested in comparing the differences between the data values of two GRIB files, you can blacklist (exclude) certain keys which show up in the comparison e.g.

```
% grib_compare file1.grib1 file2.grib1
long [binaryScaleFactor]: [-4] != [-11]
double [referenceValue]: [-1.219402587890625000e+03] !=
[-1.266985607147216796 88e+01]
        absolute diff. = 1206.73, relative diff. = 0.98961
        tolerance=0.000244141
long [N]: [2694] != [1958]
long [P]: [1385] != [712]
double [values]: 4139 out of 4160 different
 max absolute diff. = 9.1467330932617188e+01, relative diff. = 0.319192
        max diff. element 0: 1.95091751098632812500e+02
2.86559082031250000000e+02
        tolerance=0.0000000000000000e+00 packingError: [0.0313721]
[0.000244617]

% grib_compare -bbinaryScaleFactor,referenceValue,N,P file1.grib1 file2.
grib1
```

You can also **include** what you want to compare rather than **exclude** with a blacklist. E.g.

```
% grib_compare -c values file1.grib1 file2.grib1
```

Now only the values array is compared. Now you can use the -P, -R and -A flags to control the tolerances.

grib_compare: index based comparison

Suppose you have two large GRIB files and their messages are not in order, how do you compare them? One technique is to use the "-r" key for grib_compare. This will compute the md5 hash value of each message (meta-data only) and sort both files (in memory) by that md5 value before doing the comparison.
Another technique is to build **index** files from both inputs (using grib_index_build) and then compare those index files. This can be considerably faster. The user has to decide on which keys to build the index files. An example is shown here.

Here is the version with "-r":

```
% grib_compare -r $input1 $input2
```

And now using grib_index_build with MARS keys:

```
#!/bin/sh
grib_index_build -o idx1 $input1 >/dev/null
grib_index_build -o idx2 $input2 >/dev/null
grib_compare idx1 idx2
rm -f idx1 idx2  # Clean up
```

By default grib_index_build uses the MARS keys (those in the "mars" namespace). Otherwise the user can pass his/her own desired keys via the "-k" option.

An even faster version of the 2nd script can be done if we launch the two grib_index_build processes in the background (run them concurrently):

```
#!/bin/sh
grib_index_build -o idx1 $input1 >/dev/null  &
grib_index_build -o idx2 $input2 >/dev/null  &
wait  # For the background tasks for finish
```

```
grib_compare idx1 idx2
rm -f idx1 idx  # Clean up
```

How can I see the contents of an index file?

You can use the grib_dump command with the "-D" option:

```
% grib_dump -D my.index
```

And the output will look something like this:

```
--- grib_index_build: processing data/tigge_cf_ecmwf.grib2
--- grib_index_build: processing data/tigge_af_ecmwf.grib2
--- grib_index_build: keys included in the index file idx:
--- mars.date, mars.stream, mars.type, mars.step, mars.param, mars.
levtype, mars.levelist, mars.number
--- mars.date = { 20070122, 20060630, 20060623, 20060811, 20060619 }
--- mars.stream = { enfo, oper }
--- mars.type = { cf, fc }
--- mars.step = { 96, 0 }
--- mars.param = { 165, 166, 59, 260207, 156, 172, 151, 228002, 179, 3,
60, 260210, 235, 228141, 228144, 228039, 228139, 133, 189, 168, 121,
122, 167, 147, 176, 177, 134, 146 }
--- mars.levtype = { sfc, pl, pv, pt }
--- mars.levelist = { undef, 925, 2, 320 }
--- mars.number = { 0, undef }
--- 83 messages indexed
```

What are the GRIB keys numberOfDataPoints, numberOfValues and numberOfMissing?

numberOfDataPoints: This is the total number of points on the grid and includes missing as well as 'real' values

numberOfValues (=numberOfCodedValues): These two keys are the same (one is an alias for the other). This is the number of 'real' values in the field and excludes the number of missing ones

numberOfMissing: You guessed it. The number of missing values in the field

So you can write the equation:

numberOfDataPoints = numberOfCodedValues + numberOfMissing

BUFR: Performance improvement by skipping some keys

When we set the "unpack" key to decode the data section, for every Table B element key we create a set of attributes. So for example examine the output from "bufr_dump -jf" and you see meta-data attributes like:

```
"key" : "cloudCoverTotal",
"units" : "%",
"scale" : 0,
"reference" : 0,
"width" : 7
```

So in this case as well as having the key "`cloudCoverTotal`", you will also have "`cloudCoverTotal->units`" and "`cloudCoverTotal->scale`" etc.

In fact the cost of creating these extra keys is quite high so it would be advantageous for the users who do not care about these keys to omit them altogether. This can be done (from ecCodes version 2.9.0) by setting the key:

```
skipExtraKeyAttributes
```

Note: This must be done BEFORE calling unpack e.g.

```
codes_set(msgid, 'skipExtraKeyAttributes', 1)
codes_set(msgid, 'unpack', 1)
```

Now decoding will be on average 25% faster as fewer keys are created/destroyed. Note: This does not affect attributes like percentConfidence and associatedFieldSignificance. Their keys will be created if present in the message.

BUFR: How can I set keys for a subset?

You have to use the "by rank" syntax i.e. "`#n#key`". For example using the bufr_filter rules:

```
set #4#airTemperature = 3.14;
```

This sets the 4th instance of the key "airTemperature" to 3.14.
Note: Although you can use the "by condition" syntax for *getting* values of keys, this method cannot be used for *setting* such keys i.e. The following will not work:

```
# Does not work
set /subsetNumber=4/airTemperature = 3.14; ## Error
```

Also see Setting keys by rank.

How can I multiply a GRIB field values by a constant?

One way is to use grib_set with the key "scaleValuesBy". Let's say you have a field in orig.grib whose values you want to multiply by 2.1. First check the original data value statistics:

```
% grib_ls -p min,max,avg orig.grib
min          max          avg
234.554      312.085      278.977
```

You could have also checked the statistics by "grib_ls -n statistics orig.grib".
Now let's scale all values up by a factor of 2.1:

```
% grib_set -s scaleValuesBy=2.1  orig.grib  out.grib
```

And check the new values after the multiplication:

```
% grib_ls -p min,max,avg out.grib
min          max          avg
492.563      655.376      585.853
```

For adding or subtracting a constant, you can use the key "offsetValuesBy".

grib_to_netcdf: Why do I get the error "Wrong number of fields... Try using the -T option"?

You try to retrieve data from the ECMWF data archive in NetCDF format but the retrieval fails with this message:

```
ECCODES ERROR : Wrong number of fields
ECCODES ERROR : File contains 806 GRIBs, 806 left in internal
description, 745 in request
ECCODES ERROR : The fields are not considered distinct!
ECCODES ERROR : Hint: This may be due to several fields having the same
validity time.
ECCODES ERROR : Try using the -T option (Do not use time of validity)
```

This can occur if:

- you request forecast data in NetCDF format
- your data request contains overlapping "time"+"step" specifications

For example, for the ERA-Interim dataset there are two daily forecasts (00:00, 12:00), with 3-hourly forecast steps. So one could specify in a data retrieval script:

```
"date":"2016-12-01"
"type":"fc"
"time": "00:00/12:00",
"step": "3/6/9/12/15",
"format": "netcdf"
```

With the above specification you get data for the following validity times:

- time 00:00 + step 3 validity time 2016-12-01, 03:00
- time 00:00 + step 6 validity time 2016-12-01, 06:00
- time 00:00 + step 9 validity time 2016-12-01, 09:00
- time 00:00 + step 12 validity time 2016-12-01, 12:00
- time 00:00 + step 15 validity time 2016-12-01, 15:00
- time 12:00 + step 3 validity time 2016-12-01, 15:00
- time 12:00 + step 6 validity time 2016-12-01, 18:00
- time 12:00 + step 9 validity time 2016-12-01, 21:00
- time 12:00 + step 12 validity time 2016-12-02, 00:00
- time 12:00 + step 15 validity time 2016-12-02, 03:00

In this example you get *two* data values at the *same validity time* 2016-12-01, 15:00. The NetCDF format does not support multiple data values at a single time, hence the creation of the output NetCDF file fails, triggering the error message.

The solution is:

- Retrieve the data in its native GRIB format, which supports multiple data values at any one validity time.
  Then convert the data from GRIB to NetCDF format using the grib_to_netcdf tool with the **-T option**.
- Retrieve data for each forecast "time" separately, for example for ERA-Interim:
  - first with time = 00:00 and all required steps
  - then with time = 12:00 and all required steps

Is ecCodes thread-safe?

Yes, but the package needs to be built with either Pthreads (POSIX Threads) or OpenMP support. To do so, you need to make sure that one of the following options is present in your cmake configure command.

```
-DENABLE_ECCODES_THREADS=ON
-DENABLE_ECCODES_OMP_THREADS=ON
```

These options are mutually exclusive.
Note: there are known thread safety issues when GRIB multi-field support is enabled.

Python: Can I catch a specific exception?

Yes. See the full list of Python exception classes here.
Note: Client code that currently handles exceptions does not need to be changed because these exceptions are subclassed from the current `CodesInternalError`.

Building on High Performance Computer systems

Some HPC batch systems have a different hardware architecture for their login (or frontend node) to the batch node, but the frontend compilation system is targeted at the batch nodes. This is known as cross-compilation. If this is the case you may see failures in the 'make check' stage because the checks, although compiled for the backend batch nodes, are being run on the frontend nodes and therefore may not work correctly. If this is the case on your system, we recommend using a batch job to do the 'cmake; make; make check; make install' steps.

In some cases, the batch system cannot be used for compilation at all. In this case, you have to compile on the frontend but without extra flags 'configure' will assume the build is for the frontend. You can make use of the --host option to ensure the build is correct for the architecture of the batch system. Again though, the tests will fail, a small serial batch job is recommended to make sure ecCodes is installed correctly.

Note if you plan on using ecCodes in your own software that runs on the frontend nodes you will need to install ecCodes twice; one for the batch system and again for the frontend system.

If you have any questions installing ecCodes in this type of environment, please contact: openifs-support@ecmwf.int for assistance.

How do I know if a GRIB message has a Local Use Section?

The key `localUsePresent` can be used to query whether a message has the Local Use Section (this section is optional). A value of 1 means the Local Use Section is present and 0 means it is not present. E.g.

```
% grib_ls -p localUsePresent my.grib
```

How do I create my own local GRIB definitions?

See the slides in this Training course presentation.
Also the How-To Article GRIB: Converting edition 1 to 2 (The "Local configuration" section)

Can I use my own GRIB/BUFR sample files?

Yes. You can add your own samples, either to the installation directory (if you have access) or by defining the environment variable `ECCODES_SAMPLES_PATH`. The latter works like a normal Unix PATH where you have a colon separated list of directories for ecCodes to search when a sample is required.
One important requirement is that the sample file must have the extension "`.tmpl`".

To set this environment variable properly we have first to find the samples directory used by ecCodes. For this we can use the tool `codes_info` which provides some configuration information about the library:

```
% codes_info
  ecCodes Version 2.6.0
  ...
  Default SAMPLES path is used: /usr/local/apps/eccodes/2.6.0/share
/eccodes/samples
  SAMPLES path can be changed by setting ECCODES_SAMPLES_PATH
environment variable
```

Let's say you place your own sample files in `/home/eccodes/samples`. We have to set the environment variable ECCODES_SAMPLES_PATH as follows:

```
export ECCODES_SAMPLES_PATH=/home/eccodes/samples:/usr/local/apps
/eccodes/2.6.0/share/eccodes/definitions
```

Another cool trick is to use the codes_info tool itself for this purpose, rather than hard-coding the installation path with a specific version:

```
export ECCODES_SAMPLES_PATH=/home/eccodes/samples:`codes_info -s`
```

Refer to the codes_info help page for its options.

Now when you refer to a sample file (e.g. the C function `codes_bufr_handle_new_from_samples`), at run-time ecCodes will search for it in the first directory and if it is not there, it will search in the second directory (and so on)

How can I get verbose output when running tests?

When you run the tests via CMake (which actually runs ctest), you are only told if it passed or failed. After a failure you want to re-run the test and see all of what the test was doing.
So for example you run the tests first:

```
ctest
```

Now let's assume you see the test "t_bufr_ls" has failed. Now re-run just that test (not everything) with the verbose option:

```
ctest -VV -R t_bufr_ls
```

This -VV switch tells ctest not to suppress its output. Another method is to run all the tests with the "--output-on-failure" option of ctest:

```
ctest --output-on-failure
```

How can I set the verbosity level to see debug output?

There is an environment variable called **ECCODES_DEBUG**: If set to -1, it will enable brief debug-level logging messages to be displayed by the library. If set to 1, you will get very verbose output.
For example let's say we want to change the grid type of a GRIB message to be "Lambert Conformal":

```
% export ECCODES_DEBUG=-1
% grib_set -s gridType=lambert input.grib2 lamb.grib2
ECCODES DEBUG grib_set_string gridType=|lambert|
ECCODES DEBUG grib_set_long gridDefinitionTemplateNumber=30
...
```

Here the debugging output tells us setting the key "gridType" (whose type is "string") causes a low-level key "gridDefinitionTemplateNumber" (whose type is "long" i.e. integer) to be 30 (Also see http://apps.ecmwf.int/codes/grib/format/grib2/templates/3/30).

Another example showing what happens when you set the "shortName":

```
% export ECCODES_DEBUG=-1
% grib_set -s shortName=2t in.grib2 out.grib2
ECCODES DEBUG grib_set_string shortName=|2t|
ECCODES DEBUG grib_set_long discipline=0
ECCODES DEBUG grib_set_long parameterCategory=0
ECCODES DEBUG grib_set_long parameterNumber=0
ECCODES DEBUG grib_set_long typeOfFirstFixedSurface=103
ECCODES DEBUG grib_set_long scaleFactorOfFirstFixedSurface=0
ECCODES DEBUG grib_set_long scaledValueOfFirstFixedSurface=2
```

Now you can see the cascading effect of setting a key which causes others to be set. Try it when you convert from GRIB1 to GRIB2 to see how much happens behind the scenes!

Where can I find ecCode's version history?

The ecCodes version history is here.

Note: ecCodes is an evolution of GRIB-API. For the version history of GRIB-API please see here