Contents

1 Introduction to ARM Bit-Packed QC **1**

2 Notes on ARM QC Tests **2**

3 Introduction to Bit-Packing **2**

4 Reading Individual QC Test Results **3**

5 Reading Multiple QC Test Results using Masks **4**

6 Writing QC Tests as Bit-Packed Integers **5**

7 Advanced Topics **6**

8 Using QC Fields in C **7**

9 Using QC Fields in Fortran **9**

10 Using QC Fields in IDL **11**

11 Other languages **16**

1 Introduction to ARM Bit-Packed QC

In recent years, the ARM program has worked to add standardized quality control (QC) information to ARM Value Added Products (VAPs) to aid users in utilizing the VAP data and to make it easier for the Data Quality Office to assess VAPs. The fundamental idea behind the standardized QC information is that for each variable a series of QC tests are performed and the results of these tests are binary: either the data passes the test or fails it. To capture the results of these tests base level (.c1.) VAP files have bit-packed QC fields, in which each bit contains information about a particular QC test.

In this document, we describe how to use ARM bit-packed QC fields. We will explain the concepts behind bit-packing and give examples in several programming languages of how to both read and write standard ARM QC data fields. We will also provide a theoretical background on bit-packing which will allow you to extend these concepts to other languages if needed. The index at the top of the document will allow you to easily navigate to topics of interest and actual code examples.

Summary QC Data Files

To make the VAP files easier to use for the standard data user, along with the base .c1 level files, summary (.s1) files are also created for many VAPs. In the .s1 files, the results of all of the individual QC tests performed on a variable have been summarized. From the .s1 files, you can obtain information about the overall quality of the data, but do not have information about which particular tests failed. The QC fields in the .s1 data can have one of four possible values:

- 0 = Good: Data exists and passed all QC tests.
- 1 = Indeterminate: Data failed at least one "Indeterminate" QC test, but no "Bad" tests.
- 2 = Bad: Data failed at least one "Bad" QC test.
- 3 = Missing: Data is missing.

Thus, if you are only interested in the overall data quality, and do not need information on the individual QC tests, we recommend that you use the .s1 summary VAP files; in most cases you will want to accept all values with `qc_field <= 1`.

2 Notes on ARM QC Tests

Throughout this document we will use the terms "data field" and "QC field"; the former is the variable in the netCDF file that stores the actual data we are interested in (e.g. "temperature"), while the latter is the associated field that gives the QC information about that data (e.g. "qc_temperature"). For further information see the documentation on the standard ARM QC methodology.

The fundamental idea behind the ARM QC tests is that they are constructed to be binary: either the data passes the test or fails it. There is no way to store continuous state information in a standard ARM QC field; if you want to give a continuous metric (e.g. "% of potential data points used"), you will have to use Auxilliary QC fields (aqc_fields), which are not required to be bit-packed or even integer values.

Not all QC tests are designed to flag the data as "Bad"; some tests are designed simply to identify unusual circumstances in the data, or to pass information to the end user. For example, we might use one of the bits to indicate that the data was interpolated rather than directly measured. For these types of QC tests, we flag the data as "Indeterminate", and most endusers will want to use such data in their analyses.

In the ARM netCDF file, each QC field must have a number of attributes named `bit_`N`_assessment`, which describe whether the test represented by bit N is fatal or not for the data field. The possible values of the assessment are "Bad" and "Indeterminate", and provide a way to determine which bits of the QC field you would like to use to exclude data from your analysis.

3 Introduction to Bit-Packing

All standard ARM QC fields are *bit-packed integers*; this means that each bit (with a value of either "1" or "0") in the integer value contains QC information about a particular QC test. This method allows us to provide multiple QC states for a given datum in one number, but it does make the actual combined QC state of a field a little hard to discern for us humans.

For example, the first bit in the QC integer may represent the "valid_min" test; if that bit has a value of "1", that means that the data field has failed that test - i.e., it is below the minimum value allowed for that field. If that bit has a value of "0", then the data field passed that test. Thus, non-zero bits mean something unusual or noteworthy has happened in the data field.

As you know, you can represent any integer as a series of bits (i.e. as a binary number); formally, we can say that:

$$N = \sum_{b=1}^{32} w_b \, 2^{b-1}$$

for the 32 bit integer `N`, where `b` are the bits of that integer and the values `w` are either 0 or 1 for each bit `b`. This decomposition allows us to store the results of up to 32 QC tests in a single 32-bit integer.

As an example, the number 109 in 32-bit notation is:

```
0000 0000 0000 0000 0000 0000 0110 1101
```

By convention, the lowest to highest bits are read right to left, just like you would read any binary number. This is true regardless of the bit ordering in your particular machine. The methods listed in this document will still hold whether you are on a big-endian or little-endian machine.

We say that for the integer 109, bits 1, 3, 4, 6, and 7 are *set*, which means they have the value of "1". In QC terms, that means this particular data field failed tests 1, 3, 4, 6, and 7, and passed all the other tests we applied.

The basic tools we use to work with bit-packed integers are the *bitwise operators* for your programming language. A good first step would be to look up "bitwise" in your favorite language reference if you need help understanding the rest of this document.

4 Reading Individual QC Test Results

If all you are interested in is whether the data field failed *any* QC test at all, then you can simply check whether the QC field has a value of exactly zero or not. We know that if none of the bits have been set (i.e. no QC test was failed), then the QC field will have a value of zero; thus

```
qc_val != 0
```

is all we need to find out if some test or another has failed. This is, however, almost certainly not what you want to do; usually, you will still want to use data that fails at least some of the "Indeterminate" tests.

Therefore, you will need a way to determine when the data fails some *subset* of the available tests; to do that, you will have to use your language's *bitwise AND* operator.

The bitwise AND checks the corresponding bits in two numbers, and returns "1" in that bit location if *both* bits are set and "0" if either (or both) bits are not set. For example, `109 AND 43 = 41`, as we can see in this diagram (using only the lowest 8 bits for each number):

```
109:    0110 1101
43:     0010 1011
----------------- AND
41:     0010 1001
```

To determine if one particular bit is set, you need to bitwise AND the QC value with the integer corresponding to that lone "clean" bit - i.e. the integer that has that bit set to "1" and all other bits set to "0". You can, of course, hard code this clean bit integer if you want; the integer value for bit 5 will always be "16". But you can also build that clean bit integer using the *bit shift* (specifically, the *left shift*) operator.

Shifting a bit leftwards does exactly what it sounds like - it moves it one position to the left in the standard bit representation, which makes the value larger. In fact, left shifting any integer by one position is exactly the same thing as multiplying by 2; you could build a clean bit integer by multiplying the value "1" by 2 N-1 times, or by taking the exponential 2^(N-1). But bit shifting is orders of magnitude faster than those operations; bitwise operations are, in general, amongst the fastest operations on any computer.

To get a clean bit representation of bit N, you need to shift the integer "1" left by N-1 positions; to get to "16", you left shift 1 four times:

```
1:      0000 0001
----------------- left shift 4 positions
16:     0001 0000
```

Once you have the clean bit corresponding to your test, a simple bitwise AND will tell you whether that bit is set in your QC value. The following pseudocode illustrates this:

```
bit5 = left_shift(1,5-1)

if ((qc_field AND bit5) NE 0) {
   print "Bit 5 is set in qc_field"
}
```

5 Reading Multiple QC Test Results using Masks

The previous example checked just one bit; we use something like that when we are interested in finding if the data field passed one particular test, regardless of how it faired in the other tests. But the power of bit-packing lies in the fact that we can store data about *all* the tests at once - and that we can check *all* the tests we are interested in at once.

To do that we have to build a *mask* - an integer with the interesting bits set to 1 and all the other bits set to 0. Then a simple bitwise AND between our QC value and the mask tells whether any of tests fail.

For example, if we want to screen out data that fail tests 1, 2, 4, and 7, we would set our mask to 75:

```
75:     0100 1011
```

Then, a simple AND with the mask tells us what we want to know:

```
mask=75;
if ((qc_field AND mask) NE 0) {
   print "qc_field has failed test 1, 2, 4, and/or 7!"
}
```

Of course, you usually don't want to hardcode the mask to a specific integer, which won't work if the order of the tests change or if you try to reuse your code for other data. Instead, you should build a mask inside your code, by setting each bit that corresponds to an important test. This is exactly the same thing you would do to write out your own values to a QC field, and is the subject of section §6.

Using Masks to Find Good Data

Obviously, you can use a QC mask to find data that *passes* all the tests just as easily as you can to find data that *fails* one or more tests. In this case, you are looking for masked values that equal 0:

```
if ((qc_field AND mask) EQ 0) {
   // we can use this data!
   use_data_value(data);
}
```

6 Writing QC Tests as Bit-Packed Integers

Now that you know how to unpack the bits in a QC value, it is time to learn how to pack them in the first place; you will need to do this if you are going to write out your own QC fields (a requirement for new ARM datastreams), or if you are going to build a mask to compare your input QC values against.

The tool that does this is the bitwise OR, which checks the corresponding bits in two numbers and returns "1" in that bit location if *either* of the bits are set and "0" if *both* bits are not set. Therefore, `109 OR 43= 111`:

```
109:    0110 1101
43:     0010 1011
----------------- OR
111:    0110 1111
```

Coupled with an assignment operation, the bitwise OR can be used to set bits in an integer. Note that the OR is not a *toggle*; it will set the bit to 1 regardless of what it was before.

In section §3, above, we describe how any integer can be decomposed into powers of 2. From this one might infer that if you are building an integer up from individual bits, you can actually *add* the clean bit integers as an alternative to OR-ing them together. This can lead to some interesting tricks, but is a **very dangerous** habit to get into. Addition will **not** work as a substitute for the bitwise OR between any two generic integers, and if you used it for (e.g.) merging two masks together the results would be catastrophic.

7 Advanced Topics

Signed vs. Unsigned Integers

As a rule, it would be best to use unsigned integers to hold the qc values, so that you have all 32 bits available for tests. For signed integers, the leftmost bit is reserved to indicate sign; furthermore, negative numbers are usually represented in 2s complement format, so you can get some pretty weird looking values. For example, the signed integer:

```
1000 0000 0000 0000 0000 0000 0000 0001
```

is **not** -1, as you might expect, but -2147483647 (-(2^32-1)).

What this means is that if you actually have 32 QC tests, bitwise AND comparisons with signed integers might return an integer that is less than 0. Thus, you should make sure to use "not equals" rather than "greater than" zero when you check the return value of a bitwise AND, or make sure you use unsigned integers all the time.

Clearing Bits

There are a few circumstances when you might want to *clear* a bit, i.e. set the value to 0. For example, you might want to remove a test from a mask or change a QC value based on further considerations. It is also sometimes easier to build a mask by clearing bits you aren't interested in than setting ones you are.

To do this we use the bitwise NOT (also called the bitwise *complement*), which flips the value of every bit in an integer. If you take the NOT of a clean bit, you end up with a mask that has every *other* bit set to 1, and only that bit set to 0. You can then bitwise AND this with your QC value, and the result will have that bit set to 0 and all other bits untouched.

For example (again, using the lowest 8 bits for readability), let us consider bit 5, which gives an integer value of 16:

```
16:     0001 0000
NOT 16: 1110 1111
```

If we want to clear bit 5 from a QC value 59, we take the value 59 AND (NOT 16) and get the value 43:

```
59:     0011 1011
NOT 16: 1110 1111
----------------- AND
43:     0010 1011
```

Note that only the value of bit 5 has changed. It is possible to clear multiple bits at once using a mask, which is left as an exercise for the reader.

8 Using QC Fields in C

Reading QC Fields

In C, the bitwise AND operator is a single & and the left shift operator is <<. Thus, to find if a given QC field has failed test N, you need to do is something like this:

```
int N, qc_val;
...
if (qc_val & (1<<(N-1)) {
   printf("QC failed test %d\n", N);
}
```

You will probably want to hide this bitwise operation in a function or macro; for example:

```
#define qc_check(qc,bit) ((qc) & (1<<((bit)-1)))
```

will allow you to say:

```
if (qc_check(qc_val,N)) {
   printf("QC failed test %d again!\n", N);
}
```

There is a language-dependent subtlety hidden in this example. The value returned from the bitwise & operation is *not* either 1 or 0; it returns a full integer value with each bit set or not. In other words, the possible values of

```
qc_val & 16
```

are either `0` or `16`, depending on whether bit 5 was set in `qc_val` or not. In C, you can treat any non-zero integer value as a logical "TRUE", while the integer value of 0 is "FALSE". That's why the above conditional works - but it would not work in a language that had a true logical data type (like Fortran). To be completely rigorous, therefore, you should really check to see whether the value of the bitwise & is different than zero:

```
#define qc_check(qc,bit) (((qc) & (1<<(bit-1))) != 0)
```

Building a QC Mask and Setting Output QC Values

In C, the bitwise OR operation is a single |; we often use it in conjunction with assignment as |= (similar to the way we use += to add and assign in one operation).

To build a mask that checks tests 1, 2, 4, and 7, we would do something like this:

```
int mask;
...
```

```
mask=0;  // Important to set all bits to 0 to start
mask |= (1<<(1-1));
mask |= (1<<(2-1));
mask |= (1<<(4-1));
mask |= (1<<(7-1));

if ((qc_val & mask) != 0) {
   printf("QC failed an important test!\n");
}
```

Again, it is probably easier to build a function or macro to assign bits:

```
#define qc_set(qc,bit) ((qc) |= (1<<((bit)-1)))
...
qc_set(mask,1);
qc_set(mask,2);
qc_set(mask,4);
qc_set(mask,7);
....
```

Setting an output QC value uses the exact same method:

```
int output_qc=0;

for (N=1;N<ntests;N++) {
  if (fails_test(data,N)) {
     printf("Data failed test %d\n", N);
     qc_set(output_qc,N);  // or output_qc |= (1<<(N-1))
  }
}
```

Clearing QC Bits

The NOT operator in C is ~:

```
#define qc_clear(qc,bit) ((qc) &= (~(1<<((bit)-1))))
```

A quick way to build a mask that includes all possible tests is to take the bitwise complement of the integer 0. For example, this mask ignores test 3 and includes all other tests:

```
mask = (~0);
qc_clear(mask, 3);
```

QC Tests for VAPS

In general, the QC tests occur in the middle of your VAP code, as you calculate a value and then check to see if is reasonable. For this reason, you usually want to put the bits associated with each test in a variable, to make sure you keep them straight and to help the readability of your code. For example:

```
static int QC_VALID_MIN=1;
static int QC_VALID_MAX=2;
static int QC_INTERPOLATE=3;
...

val[i]=calculate_value(data, &interpolated);

if (val[i] < valid_min) {
  qc_set(qc_val[i], QC_VALID_MIN);
}

if (val[i] > valid_min) {
  qc_set(qc_val[i], QC_VALID_MAX);
}

if (interpolated) {
  qc_set(qc_val[i], QC_INTERPOLATE);
}
```

9 Using QC Fields in Fortran

Reading QC Fields

In Fortran 90 (I'm not sure about earlier versions), you do the same thing we did in C: left shift the integer "1" N-1 positions and do a bitwise AND with your qc value. The Fortran operators for this are `IAND` and `ISHFT`:

```
if (IAND(qc_val,ISHFT(1,N-1)) .ne. 0) then
  write(*,*) 'Bit ', N, ' is set in qc_val
endif
```

However, for comparisons with clean bit integers, Fortran actually has an simpler logical function called `BTEST`:

```
if (BTEST(qc_val,N-1)) then
  write(*,*) 'Bit ', N, ' is set in qc_val
endif
```

`BTEST` takes as its second argument the bit position of the clean bit integer we want, rather than the integer itself. (Note that, unlike most things in Fortran, the bit position is 0-offset, so you still have use bit position `N-1` when you mean bit `N`.) For QC purposes, `BTEST` is asking exactly the same question as we are: is bit N set in our QC value?

Using Masks in Fortran

Unfortunately, the BTEST function will not work with a mask, as it is designed to check one bit at a time. Therefore, you will have to use bitwise AND explicitly on your qc_val and the mask:

```
if (IAND(qc_val,mask) .ne. 0) then
  write(*,*) 'QC_val failed some important test'
endif
```

Building a QC Mask and Setting Output QC Values

In Fortran, the bitwise OR is called IOR; there is also a function called IBSET which sets the bits explicitly without having to do any left shifting. Thus, to build our mask for tests 1, 2, 4, and 7, either one of these examples will work:

```
! easy way
mask=0
mask=IBSET(mask,1-1)
mask=IBSET(mask,2-1)
mask=IBSET(mask,4-1)
mask=IBSET(mask,7-1)

! now do it the hard way, with bitwise ors
omask=0
omask=IOR(omask,ISHFT(1,1-1))
omask=IOR(omask,ISHFT(1,2-1))
omask=IOR(omask,ISHFT(1,4-1))
omask=IOR(omask,ISHFT(1,7-1))

if (IAND(qc_val,mask) .ne. 0) {
   write(*,*) "qc_val failed an important test"
}
```

Again, building an output QC value is similar:

```
output_qc=0;

do N=1,ntests
   if (failed_test(data,N)) then
      write(*,*) "data failed test", N
      output_qc=IBSET(output_qc,N-1)
   endif
enddo
```

Clearing QC Bits

The Fortran bitwise complement is called `NOT`, but there is also an `IBCLR` function which clears a bit directly in much the same way as `IBSET`:

```
mask=IAND(mask,NOT(ISHFT(1,bit-1)))
mask=IBCLR(mask,bit-1)  ! exactly the same thing
```

10 Using QC Fields in IDL

Reading QC fields

The IDL bitwise AND is called `AND`; the bit shift operator is called `ISHFT`. Thus, IDL code will look like this:

```
if ((qc_val AND ISHFT(1UL,N-1)) ne 0) then begin
  print, "Bit ", N, " is set in qc_val"
endif
```

Note once again that the output of the bitwise `AND` is not a logical but an integer, and thus you must do the `ne 0` comparison to use it in a conditional. (Also, make sure you have enclosed the bitwise `AND` in parentheses before doing the comparison, as the `ne` operator binds tighter than `AND`, for some crazy reason.)

Building a QC Mask and Setting Output QC Values

The IDL bitwise OR is called `OR`, and so we could build our mask like this:

```
mask=0UL
mask=mask OR ISHFT(1UL,1-1)
mask=mask OR ISHFT(1UL,4-1)
...
```

Here is a function that will build a mask for you; the argument `bits` is an IDL integer array with the (1-offset) positions of the bits you want set:

```
FUNCTION build_mask,bits
  mask=0UL
  FOR i=0,n_elements(bits)-1 DO BEGIN
    mask = (mask OR ishft(1UL,bits[i]-1))
  ENDFOR
  return, mask
END
```

(but see section §10, below, for a slicker way to do this). You would call this function like this:

```
x=[1,2,4,7]
mask=build_mask(x)

if ((qc_val AND mask) ne 0) then begin
  print, "Qc_val failed tests 1, 2, 4, and/or 7!"
endif
```

Clearing QC Bits

The IDL bitwise complement is called `NOT`:

```
mask = mask AND (NOT ishft(1UL,bit-1))
```

will set `bit` in `mask` to "0".

32 Bit Integers in IDL

Most languages have a default integer type of at least 32 bits, but for IDL "integer" means "16-bit signed integer". Thus, you should get in the habit of casting all the integers used for QC values and masks as "unsigned long", which give you the full 32 bits:

```
mask=0UL
...
clean=ISHFT(1UL,bit-1)
```

Most IDL functions and operations return a value of the same type as the most rigid argument, so adding or OR-ing integers of any type to `mask=0UL` will return a type of "ulong", and the variable `clean` above is also an "ulong". But you have to be careful; some functions return a different type (e.g. `total` returns a floating point number regardless of the type of its inputs), so you have to explicitly cast it back using `ULONG()`:

```
mask=ULONG(total(ISHFT(1UL,bits-1)))
```

In later versions of IDL there is even a 64 bit unsigned integer type (called "unsigned long long"), which may be prudent to use if we ever have 32 or more QC Tests to perform:

```
mask=ULONG64(total(ISHFT(1ULL,bits-1)))
```

Vector Processing and Programming Hints

The bitwise operations in IDL are vectorized, which means you use them to avoid looping and can easily construct a logical vector that screens out the bad data from your arrays, using the `where` command. For example:

```
fid = ncdf_open(file)
ncdf_varget, fid, 'time', time
ncdf_varget, fid, 'temperature', temp
ncdf_varget, fid, 'qc_temperature', qc_temp
...

x=[1,2,3,5]  ! will want to scan bit_N_assessment attributes
             ! in a real application
mask=build_mask(x)

good_index=where((qc_temp AND mask) eq 0, ng)
bad_index=where((qc_temp AND mask) ne 0, nb)

!! now use only the good points
plot, time[good_index], temp[good_index]
mean_daily_temp=mean(time[good_index])

print, "The percentage of good points in this file is", float(ng)/float(ng+nb)
...
```

The `build_mask` function given in section §10 can actually be simplified by removing the loop in favor of vector operations:

```
FUNCTION build_mask,bits
  return, ulong(total(ishft(1UL,bits-1)))
END
```

(Note the use of `total` to sum the clean bit integers; we have warned you about this, so make sure you know what you are doing before you try a stunt like this.)

If you had a string array `qc_assess` that held the QC assessment string "Bad" or "Indeterminate" in each slot (i.e. `qc_assess[2]` would tell us whether QC Test 3 was "Bad" or "Indeterminate"), then:

```
mask=ulong(total(ishft(1UL,where(qc_assess eq "Bad"))))
```

would build our QC mask. (Note that we do not have a "-1" on the second argument to `ishft`, as IDL arrays are already 0-offset). This works even if none of the assessments is "Bad" - in that case the `where` returns -1, and *right-shifting* the integer "1" one position gives you the integer "0".

2D data and Vector Processing

It is tricky to use IDL vector processing with multi-dimensional data, as it collapses the multiple dimensions down into a big 1D array. For example:

```
        ncdf_varget, fid, 'temperature', temp2D
        ncdf_varget, fid, 'qc_temperature', qc_temp2D
        mask=build_mask(x)
        good_index=where((qc_temp2D AND mask) eq 0, ng)
```

returns `good_index` as a big 1D array of size (N*M)-nbad, assuming `temp2D` is dimensioned as an NxM array. If you use `good_index` to subset `temp2D`, you will get a 1D array out:

```
        IDL> help, temp2D[good_index]
        <Expression>    FLOAT     = Array[770400]
```

although you can use the index to assign data without changing the form:

```
        IDL> temp2D[bad_index]=-9999.
        IDL> help, temp2D
        TEMP2D              FLOAT     = Array[535, 1440]
```

In some cases it will be best to loop over all but one index, saving the vector processing for the final index. This example condenses the 2D QC data down to 1D QC, with a single QC value for each sample time:

```
        qc_temp_1D=replicate(0UL, n_elements(time))
        for i=0, n_elements(levels)-1 do qc_temp_1D = qc_temp_1D OR qc_rh[*,i]
```

In this case, our 1D QC has a bit set if *any* value in the profile has that bit set.

Building A Mask in IDL from an ARM NetCDF File

IDL is surprisingly annoying to use in certain situations; the netCDF functions are bulky to use and often require more lines than one might think. Furthermore, they are not vectorized, so one cannot get around the inevitable looping. Nevertheless, it is possible to build a function to read a netCDF file that uses ARM-standard QC conventions and return a QC mask for all "Bad" QC tests. This is an example of such a function:

```
        FUNCTION ncdf_get_mask, fid, name, GLOBAL=global
         mask=0UL
         val=''

         IF keyword_set(global) THEN BEGIN
           inq=ncdf_inquire(fid)
           natts=inq.ngatts
         ENDIF ELSE BEGIN
           inq=ncdf_varinq(fid,name)
           natts=inq.natts
         ENDELSE
```

```
 FOR i = 0, natts-1 DO BEGIN
   IF keyword_set(global) THEN BEGIN
     attname=ncdf_attname(fid, /global, i)
   ENDIF ELSE BEGIN
     attname=ncdf_attname(fid, name, i)
   ENDELSE

   ;; use regular expressions; could also use strmid or something
   bar=stregex(attname,'bit_([0-9]+)_assessment', /subexpr, /extract)
   IF (strlen(bar[1]) GT 0) THEN BEGIN
     IF keyword_set(global) THEN BEGIN
       ncdf_attget,fid, /global, attname, val
     ENDIF ELSE BEGIN
       ncdf_attget,fid, name, attname, val
     ENDELSE

     IF (string(val) EQ "Bad") THEN BEGIN
       b=fix(bar[1])
       mask=mask OR ishft(1UL,b-1)
     ENDIF
   ENDIF
 ENDFOR
 return, mask
END
```

An example of this function in use is:

```
fid=ncdf_open(file)
ncdf_varget, fid, "time", time
ncdf_varget, fid, "level", level
ncdf_varget, fid, "watervapor_rh_level", rh
ncdf_varget, fid, "qc_watervapor_rh_level", qc_rh

;; this returns a mask built from the global attributes
;; qc_bit_N_assessment
global_mask=ncdf_get_mask(fid, /global)

;; This returns a mask built from field level attributes
;; It will properly return 0 if no field level assessments
;; are given, which means we should use the global ones instead
rh_mask=ncdf_get_mask(fid, "qc_watervapor_rh_level")

;; This defaults rh_mask back to the global mask
if (rh_mask eq 0) then rh_mask=global_mask

ncdf_close, fid
```

```
;; RH is 2D, so collapse QC down to one value per profile
qc_rh_1D=replicate(0UL, n_elements(time))
for i=0, n_elements(levels)-1 do qc_rh_1D = qc_rh_1D OR qc_rh[*,i]

;; find profiles that do not have any bad data
gdx=where((qc_rh_1D AND mask) eq 0, ng)

;; plot good surface values of rh
if (ng gt 0) then plot, time[gdx], rh[gdx,0]
```

11 Other languages

The bitwise operators in most other languages (at least the ones I have looked up) tend to follow the C syntax: `&` for AND, | for OR, << for left-shift, and ˜ for NOT. This includes perl, java, python, and ruby - although, as always, I encourage you to consult the language's documentation for details, as there are often interesting subtleties in how things are implemented in different langauges.

Matlab has it's own bitwise functions: `bitand`, `bitor`, `bitshift`, and, like Fortran, the direct functions `bitset` and `bitget`. Note that the bit position in these function is 1-offset rather than 0-offset, so using `bitset` or `bitget` you should use `N` rather than `N-1`, although you will still have to use `N-1` with `bitshift`.