RTP Implementation Notes

Howard E. Motteler

July 9, 2001

Abstract

This is a collection and catch-all for stuff that didn't belong in the user's guide or README; it is mostly a discussion of both HDF 4 and more general RTP implementation issues.

1 Introduction

While the Fortran RTP interface is (hopefully) relatively user-friendly, the Fortran implementation is complicated in part because it tries to be sensible about variable field sets and field sizes; this is convenient for the user, but is the sort of thing that is much easier from a high level language like Matlab, than from Fortran or C. A simpler implementation might map Fortran structures directly to the HDF vdatas buffer, but this would require recompilation for every format or field variation, resulting in a succession of incompatible file formats, and would need to include space for radiances and observations even when they weren't used.

2 The Fortran/C API

The Fortran/C implementation matches up field subsets from the API's static reord structures and the HDF 4 vdata field sets in a relatively efficient way.

It builds a common field subset and then builds a set of header and profile field pointers and lengths used for subsequent buffer copying. Since more than one RTP file might be open at a time, more than one set of pointers must be kept track of, thus the RTP channels are needed.

The Fortran interface procedures are mainly just wrappers for lower-level C routines, except for rtpopen.f, which moves attribute data between static and dynamic structures.

Note that when changing or adding profile or header fields or field sizes, the C structures and ascii field lists must be kept in exact correspondence; also the C and Fortran structures and parameters must match exactly.

rtpwrite1 starts with field lists for the static Fortran header and profile structures, and uses values in the header size fields (nrho, nemis, ngas, mlevs, nchan, and pfields) to build new, generally more compact, field lists for the header and profile vdata buffers. The structure and vdata buffer field lists are matched up in pvmatch, which also generates buffer pointers. These pointers, the vdata buffers, and some related info are saved in the RTP channel records. rtpwrite1 then writes the header data, and the header and profile attributes, and returns an RTP channel on which the profiles may be written by rtpwrite2.

The "pv*.c" routines are fairly generic, and implement C-structure interface to HDF vdatas. The rtp*.c (and pvmatch.c) routines do the field matching, buffer mapping and other RTP-specific stuff. By changing just the pv layer, it should be possible to move the existing RTP API to a different underlying representation; for example RTP might be implemented directly as binary files, writing flists to the file to keep track of things, or might be implemented with HDF 5.

3 General HDF 4 Implementation Issues

HDF is a plausible choice for representing atmospheric profiles; it is portable, and can be used in such a way that the data files are largely self-documenting. Test implementations of RTP were done as HDF 4 "vgroups" of SDS's with one group per profile, and as HDF 4 "vdatas", with one vdata for the header, and one for the profile array. (The HDF 4 SDS format is the basis of the EOS swath format, and the Vdata format the basis of EOS point; the RTP vdata setup is similar to an EOS point file with two EOS-point "levels", one for the header and one for the profile data.)

The first implementation, based on vgroups of SDSs, was more flexible, as different profiles can have different level sets, with no fill values needed, and new fields can be added–e.g., radiances to a profile set–as proper updates, without having to rewrite the whole file. Unfortunately, limitations of HDF 4–both default compile-time parameters, and also issues of efficiency–restricted the total number of profiles representable in this way, in a single file, to on the order of 500.

The second implementation, based on HDF 4 vdatas, allows for much larger files, up to 2 Gbytes with no real limit on the number of profiles, up to the overall size limit, with the restriction that at least within a particular file, the field set and field sizes are fixed, though these can both vary between files. Appending profiles with the same field set works fine with this representation, but adding new fields requires reading, updating, and then rewriting the file. While such limitations are irritating, overall, for HDF4 this is a more practical approach than using vgroups.

A third possible HDF4 implementation, that I didn't actually try, would be to take an SDS (i.e. a simple array) based implementation for a single profile and extend the last (or first, for the row-oriented) dimension of all the arrays with a profile index. A significant limitation of this approach is that only one of the arrays/fields can be extended once the file is created, and it was not pursued further.

a language-independent representation of proper structures and arrays of structures would be preferable to any of the HDF 4 schemes; HDF 5 appears to provide something like this, but was not available in time for this work.