df: A Proposed Data Format Standard

Leslie R. Lait Eric R. Nash Paul A. Newman

January 31, 2001

Contents

Chapter 1

Introduction

One of the underappreciated benefits of computer technology in scientific research is the exchange of data among different research groups. The ability of relatively small physical devices to store and retrieve vast amounts of information has opened up new methods of collaboration between scientists which were unthinkable in the days of personal journals stuffed with long lists of hand-written numbers. We believe that the cross-fertilization of ideas engendered by these new capabilities can only aid scientific progress.

Too often, however, the technology is not used to best advantage. Typically, a researcher will mail to a colleague a reel of magnetic tape or a diskette with a data file written on it, accompanied by a photostatic copy of a document outlining the file format. If the recipient is fortunate, a sample program to read the data will be included as well. More often than not, however, the document will be obsolete and fail to reflect the actual format of the data. Also, the sample program will be filled with mysterious local conventions and unusual subroutines—crucial to successful operation—which are unknown outside the colleague's site.

If the recipient, through the application of clever graduate students, finally manages to crack the code on this digital Rosetta Stone, he is likely to get requests from others of his colleagues for the data (with the permission of the data's originator, of course). Many of the caveats attached to the data by the first scientist will have been forgotten by this time and hence will fail to be passed along to these third parties, for whom the warnings may have critical relevance.

Add to these difficulties the more computer-specific problems of incompatible media, binary floating-point representation, machine word size, and so forth, and one has a significant obstacle to data exchange.

The use of a standard file format can reduce or eliminate many of these problems. By definition, a standard format can be read by a diverse group of machines and people. In addition, a properly designed format will contain enough self-documentation that the other scientists can not only read the data, but make some sense of it as well.

Several such standards have been promoted in recent years [NCSA Software Tools Group, 1989], [Rew, 1990], addressing varying tradeoffs between the conflicting needs and goals of a standard format. Some have emphasized, for example, portability of data over computer networks to the detriment of all other considerations. Others have insisted that the entire format, especially all metadata, be in what they consider "human readable" form. Still others have attempted to impose a particular paradigm upon the data; users are expected to think of their data in the ways that the format designers demand.

Unfortunately, the tradeoffs imposed upon these formats by their creators often violate scientists's needs. The form that the data files take should be driven by the needs of the research, and those needs vary from group to group. Computer programmers are not physical scientists, moreover, and frequently misunderstand what those scientists require. What is needed is a data format which will leave strategic decisions and paradigm choices in the hands of those who will actually be using the data sets.

The file format proposed herein is intended to satisfy this requirement. Its creators are practicing scientists, rather than computer professionals, and thus it was designed at every step to address the actual needs of working scientists, rather than the mere perception of those needs by computer programmers. This format imposes no paradigm, no grand stratagem for data management. Rather, it seeks to provide a common language to allow scientists to express their data in the way they think is best. As much as possible, strategic decisions and tradeoffs are left in the hands of the users, since the wisdom of each such decision depends upon a user's specific application.

We term this format "df", and this document describes it. This first chapter discusses some of the issues involved in choosing or designing a standard file format; programmers only interested in implementing the df format itself may wish to skip to the next chapter. Subsequent chapters define the format, give examples to clarify some of the concepts introduced, and discuss how well the df format meets its design criteria. Finally, a set of appendices is included to aid in implementing software to read and write this format. The second and later chapters are written primarily for use by programmers; scientists and managers may find them uninteresting.

1.1 Issues to be Addressed

In trying to create a new file format, several sets of conflicting and mutually contradictory goals must somehow be reconciled.

For example, the files should be completely self-documenting, but that entails writing all sorts of information which will cause the file size to swell. On the other hand, the data should be compact, since there will be many files, and it is desirable to conserve disk space.

One would like the data to be portable, so that files can be moved from machine to machine without having to convert the data. However, reading the files ought to be efficient, to minimize run time and CPU charges.

Ideally, the new data files would adhere to some sort of universal standard. Individual projects and users, though, should be able to customize aspects of the files to meet their special needs.

Any file format specification reflects a tradeoff or compromise between these (and possibly other) sets of mutually contradictory goals. We wish to find the file format whose compromises best meet our needs.

1.1.1 Towards a Standard Format

One should begin by asking the fundamental question, "Why use a standard format?" What does one gain by using such a format over the special-purpose, machine-specific formats commonly used in the past?

Three major reasons come to mind. The first is portability; a standard format makes it easier to move datasets to the new computers one acquires over the years, as well as to machines at collaborating sites. The second reason to use a standard format is so that the contents of a strange file can more readily be understood. Such a file may not necessarily be from a foreign site; one's own datasets, inspected years after the original programmers and project scientists have passed into the dust of history, can seem very unfamiliar. The third reason is to avoid re-inventing a new format each time a new dataset appears: using a standard format allows greater re-use of software to read and write data and also permits the construction of a set of tools which can be applied to a wide variety of data. A standard format, if it is to be useful, must therefore possess certain qualities which fulfill these three basic needs: portability, understandability, and reusability.

Portability

First, the format must be portable between machines. By "portable," we mean simply that one should be able to read the file, not necessarily make sense of its contents (that falls under "understandability," discussed below).

Ideally, portability would require that the dataset be written in terms of some explicitly portable binary data representation, such as the eXternal Data Representation (XDR) promoted by Sun Microsystems, Inc. Data could then be moved from machine to machine without any need for conversion.

Even if one relaxes this condition and permits native binary representations, though, portability also requires that a standard format cannot rely on special file types or file record formats which are unique to a single type of machine. That is, the concepts behind data structures must be uniform across machines, even if the implementation of those concepts is permitted to vary from machine to machine. For example, all machines used in scientific data processing have integer and floating-point data types, however differently they may be implemented. Few machines, on the other hand, have special representations for irrational numbers; one should then exclude irrational numbers from the data types permitted in the file.

In fact, because of the wide variation in the capabilities of machine operating systems (from personal computers to supercomputers), portability considerations require that any data file format specification be limited to the information written into a file, and not concern itself with how files are named or how they are arranged and manipulated within a data system. The format should not even depend on the concept of "file;" it should be possible to implement it using any sort of bit stream.

Understandability

Secondly, a standard format must in some sense be understandable, or self-describing. To begin with, a file should always identify itself as conforming to the standard—the user must somehow always be able to test simply whether a given file was written in the format. One should also be able to determine the binary representation used in the file, whether it was written in a given machine's native binary data representation or whether it was written in a portable representation such as XDR. For example, if a site using a Unix workstation obtains a dataset from an IBM mainframe, the Unix site should be able to determine if the IBM file is in the standard format, as well as what binary data representation was used (the IBM site might have obtained the dataset from a Cray site, after all).

A standard format should also be self-documenting in the more usual sense: metadata (data about the data) should be contained within the file. This documentation must include some form of identification of the data in the file as well as its units, dimensional structure, and any special processing notes and/or comments of which users of the data should take note. (Some of these notes will apply only to a subset of the data—provision must be made for specifying such subsets.) Additionally, bad or missing data points in a regular field must somehow be flagged.

In addition, it would be most useful to have an audit trail mechanism to maintain a sort of "family tree" detailing the lineage of a dataset which has been derived from other datasets. Thus, for example, the output from model runs whose initializations were obtained from various data files can be identified clearly.

Reusability

By definition, a standard file format relieves the user of the necessity of designing a new format for each new dataset. In addition, software which reads or writes the standard format is easily re-used for each new dataset created. This makes much more efficient use of scientists' and programmers' time: instead of learning the detailed bit structure of many different formats, they need only learn (and implement software for) one.

1.1.2 Away from a Standard Format

Having examined briefly the advantages of standard formats (as well as the attributes a format must possess to enhance those advantages), one must next consider some of the disadvantages of standard formats, in order to select a design which minimizes them.

Inflexibility

The fundamental difference between a standard and a custom format, of course, is who specifies the format of a file. With a standard, the defining authority specifies the format for everyone; without a standard, each user is free to design his or her own. Of course, whenever a user lets someone else design a format for his own data, he runs a very real risk of losing capabilities he needs to accomplish his task. That is, it is possible for a standard specification to be too complete, too narrowly defined. And if it is so tightly defined that it fails to provide the flexibility needed by the user, then the standard will be ignored by that user. (Standards which are ignored by their users soon cease to be standards.)

But, how tight is too tight? How much flexibility is too much? The answer lies in the ability of the user to make his or her own choices about what information can be incorporated into the file.

In writing a data file, a host of tradeoffs and compromises have to be made. If a standard file format is to meet the unique needs of its users, then they must to a large extent be able to make their own strategic choices between various conflicting goals. Thus, only a basic core format should be specified by a central defining authority, within which the user should have freedom to specify how the data are to be represented.

That is to say, a user must be able to say anything she needs to say, but all users should speak the same"language" (file format).

Another, related, issue is whether a user can express any data needed in the standard format. Modelling a data field as a simple array of variables may not be the most appropriate approach in every case. One can perhaps think of each datum as a cluster of component items located in some coordinate space; the structure of such a cluster—its dimensions—constitute what we call "Level 0" dimensions. Examples might be the components of a wind stress tensor or a wind vector. Each datum is also associated with a set of coordinates specifying its location; we can split these dimensions up into those which vary within a data record in the file and those which vary between data records. We call these "Level 1" and "Level 2" dimensions, respectively. Finally, there are those coordinates at which the data are not located, but over which the data have been averaged in some way; these we call "Level 3" dimensions. Any standard file format must distinguish between and allow for the specification of any or all of these various levels of dimensions.

A user must also be able to label or flag any arbitrary subset of the data with an informational tag of some sort.

High Overhead

Another disadvantage associated with standard file formats is the overhead involved in reading and writing them. Data written in a portable binary representation must be converted to a machine's native representation before calculations can be performed. The metadata for self-documentation eats up valuable disk space. As a result, users in search of high storage density and performance end up avoiding standard formats.

The binary representation issue can be dealt with by allowing native machine-dependent formats. In this case, however, the file must still contain information identifying the file as being in the standard format written in a particular native representation, and that information must still be readable from any machine. Recalling our previous example, a Unix system, having retrieved a file from an IBM mainframe, should be able to tell that the file is written in the standard format using Cray native binary representations.

It is reasonable, then, not to demand that a standard format exhibit strict binary compatibility across machines: if necessary, one can convert one machine's native binary format to another's. What is important is to ensure that one knows the type and structure of the data one is converting, and that the data types being converted (as opposed to the implementation or representation of those data types) not be proprietary to a single machine. The use of a file format standard satisfies the first condition; using only portable data types satisfies the second. Types such as characters, bytes, signed and unsigned integers, and single and double precision floating-point meet this criterion. Complicated structures and pointers do not.

Allowing native binary representations in the dataset format will eliminate the guarantee of being able to read a dataset fresh off a network, but with an extra binary representation conversion step added, the overall goal of being able to port the dataset easily to new machines is met. Furthermore, for those applications in which true "out of the box" readability is required, use of a portable representation should still be allowed.

Another source of computational overhead, aside from data conversion, is requiring the user to use a standard software package. Of course, standard packages are, in themselves, good and useful. They encourage people to read and write the standard format who do not wish to expend the time or effort to understand the format and create their own readers and writers. Nevertheless, no set of subroutines can operate at maximum efficiency in all applications. If users require high performance, they should be able to consult the standard specification and write their own I/O software. Documenting the reading and writing subroutines only, while keeping the underlying file format secret, prevents these users from adopting the standard.

In addition to computation overhead, standard file formats are often associated with greatly increased disk storage requirements. Aside from the increased storage requirements of certain portable binary representations, information for self-documentation accounts for much of this file-size bloat.

Users manipulating a handful of toy datasets on their personal computers may be able to live with the larger file sizes. On a large supercomputing system, however, with ten years or more of daily global meteorological data, the extra disk space required can be prohibitive.

How can one squeeze the files down?

Metadata written as plain text generally requires much more storage than that written as, say, integer codes. In seeking to reduce this overhead, then, one must examine the question of whether the metadata should be machine-readable or human-readable.

While some may claim that metadata must exist in humanly-readable form, it should be pointed out that the only storage media which allow humans to read the data directly are punched cards and paper tape, neither of which is used in today's research environment. Most commonly, data are stored as microscopic magnetic or optical patterns in some special material, and a computer is required to read them. Thus, the question becomes not, "Shall the metadata be human-readable?" but rather "What kind of computer tool shall be used to inspect the metadata?"

The case for making metadata in "human-readable" form is that a simple text editor can be used to inspect it, and everybody has a text editor. Unfortunately, many editors will choke on binary data which may be found elsewhere in the file (and if the entire file, including all data, is written as text, then many editors will still be unable to hold the entire file in memory). Therefore some specialized tool must be written and distributed which will allow one to browse or inspect a dataset. As long as such a tool must be used, however, it really does not matter how human-readable the dataset is; what matters is how human-readable the tool's output is.

Furthermore, metadata encoded as text comments embedded in a data file tend to be exclusive towards a single language, such as English; this can be a disadvantage when dealing with an international community.

The case for machine-readability, on the other hand, can be made by considering the direction in which scientific data processing is moving. In the past, many datasets were relatively small—small enough for one person to comprehend entirely. These were the sorts of datasets which had been found in laboratory notebooks, and storing and processing such datasets on computers was mainly a matter of convenience. With the advent of satellite data and output from sophisticated numerical models, the volume of datasets has grown, so that computer processing has become necessary rather than merely convenient; no one person could hope to perform calculations by hand on such datasets. In the future, one should expect more of the "grunt work" to be taken over by computer tools of increasing sophistication and speed. It would take the advent of true artificial intelligence for machines to do science—no foreseeable machine will be able to detect trends in atmospheric constituents, for example—but more of the routine, mechanical work (such as "Find the global temperatures on the 50-millibar pressure surface for 12 January 1989, and plot them up for me.") will be done by computers. It is profitable therefore to make the metadata relatively simple for a computer to understand. Naturally, this has to be limited in some respects; careful explanations of unusual experimental conditions are best described in human-readable terms, since they tend to be unique and difficult to encode. But such common items as data identifications, units, and dimensions should be machine-readable.

Choices made on this issue, of course, affect those made on others. If a special machine-readable code is used for, say, data identification, then one must also decide whether the data ID codes are defined by a central authority or by each user. Additionally, one must decide where to put the machine-to-human translations of the data ID codes. Again, a reasonable compromise is for a basic framework to be set up by a central authority, with the ability for users to fit their own code definitions into that framework. In this way, even if a code definition used in a foreign dataset is missing from a site's own list of code definitions, a clever utility program could make some sense of what the code means.

Another way to make files smaller would be to allow for packing and compression of data. By "packing," we mean the scaling of floating-point data to smaller integers occupying less storage. By "compression," we mean the reduction of redundancy in a bit stream by such algorithms as Lempel-Zif. The format should probably allow only a few standard methods, and, once the method is specified by the user, the data should be packed and/or compressed transparently by the writer subroutines (and, of course, uncompressed and unpacked transparently by the reader subroutines).

Complexity

Physical scientists as a class have long demonstrated a propensity for choosing short-term benefits over long-term considerations. Thus, they frequently prefer to dump their data arrays straight from their analysis programs with little or no thought of portability or understandability; it is just simpler to do it that way.

When a standard format is used, though, an added degree of complexity is introduced. Data and metadata have to be written out in a certain order with a certain structure. The more general and flexible a standard is, the more complicated writing its datasets will be, since the structures of the data must somehow be described instead of being assumed. Even if the format is hidden behind a standard library of I/O routines, learning to call those routines (or even linking with the appropriate libraries!) represents an obstacle for many scientists.

Limited Languages/Systems

As mentioned above, some standard formats hide the actual format of the file and force the user to use a standard software package to access it. In addition to the performance problems this entails, the user is also left dependent upon the efforts of the group writing the package to create a version for his or her language or computer system of choice.

This group usually takes the position, "With our limited resources, we cannot support every language/machine in existence, and there is just not enough demand for the version you want." Of course, without a version for that language or machine, demand stays low, and the standard format remains unused.

How Do We Know it is Standard?

In some formats, the file format itself is the only thing specified, and users are free to write their own software to read and write the data files. This allows customization and streamlining, but it also opens up the possibility that the user will get it wrong, that the files thus produced will advertise themselves as being in the format when in fact, they do not conform.

This problem can be substantially reduced, however, by the use of some sort of checkout utility to verify a newly created file.

1.1.3 Reconciliation

We have seen that standard file formats have both advantages and disadvantages. By choosing format characteristics which maximize the advantages and minimize the disadvantages, one can arrive at a format which is usable in almost all circumstances.

The trouble is, though, that many of the advantages (such as self-documentation) are in direct conflict with reducing the disadvantages (such as reducing the file sizes). One must therefore choose some sort of tradeoff between conflicting goals.

It is tempting under these circumstances to simply choose a tradeoff somewhat arbitrarily and force it upon the users. Many users—those whose needs the format does not meet—will walk away. Worse, others will design a different format based on a different set of tradeoffs, and one ends up with a whole series of (incompatible) "standards." This is exactly the situation which scientists face today.

Several scientific communities have proposed their own standard formats [Ramirez, 1991], [Pullen, 1990], but for wider interaction and exchange of data between disciplines, a more general format is needed.

If a standard format is to be successful, it must be used on a large number and a wide variety of datasets. One should not restrict its use to a small set of toy-like data files viewed and manipulated on personal computers (PCs) only. Likewise, it would be foolish to design a format to work only on massive databases on large mainframe computers. Rather, we need a format which is usable across the board, since files from PCs will be submitted to supercomputers, and small pieces of supercomputer datasets will be split off and sent to individuals using PCs.

The standard format must be flexible enough, then, for users to make their own strategic choices between efficiency and portability, between self-documentation and file size, and between the ease of using standard I/O subroutines and the performance increase gained with custom software. A high degree of user-specifiability is necessary, then, within a basic core format, or framework, specified by a central authority. This includes the ability for a user to include any data which needs to be included.

Again, a user must be able to say anything he or she needs to say, but all users should speak the same “language.”

1.2 The “df” format

The creators of this proposed standard did not lightly reject the other formats. The time and effort in designing a standard format is substantial, and they did everything in their power to avoid it. Nevertheless, they have reluctantly concluded that the existing formats which they have encountered do not meet the criteria that they believe are essential for a truly useful standard format for scientific data. This is unfortunate, since it adds yet another proposed standard to the growing list. Yet, if our needs as scientists are to be met, it seems necessary.

This is not to say that the proposed format, which is called “df,” is the final word on the subject. While considerable thought went into implementation and efficiency issues, this standard will undoubtably have some aspects whose implementation will make computer professionals groan, and certain constructs may not be as elegant or efficient as those designed by computer scientists. This standard should be viewed as an object lesson or testbed for ideas, detailing some of the features scientists need in a file format. If these features are incorporated into whatever format wins out in the end, the effort devoted to this standard shall have been worthwhile.

The authors believe that good software and good standards should be derived from a careful consideration of users’ needs—not programmers’ needs, not programmers’ ideas of users’ needs, but the users’ themselves. Scientists should be consulted from the beginning, not near the end, of the design process, and that consultation should continue through to the end. Their real, perceived needs—in what ways do they think of their data? what sort of premium do they place on disk space?—must be given paramount importance. One does not begin by asking, “what do you think of this data format?” One should begin by observing scientists as they collect, manipulate, and analyze their data; from these observations, a clearer picture of their needs can be obtained. (And an afternoon’s interview will *not* suffice!) We believe that flexibility and user-specifiability must be built-in from the ground up, not tacked on as an afterthought.

The file format proposed herein strikes a balance between the conflicting demands outlined above. Users and/or projects are allowed to determine their own balance between efficiency and portability, and yet a single standard is adhered to in either extreme.

For example, provision is made for portability of data where desired, but where efficiency is preferred, machine-specific representations are used (and are identified as such in the file). This standard also allows for various compression and data-packing methods to conserve storage, again at the discretion of the user. Bad or missing data are provided for, and other sorts of processing notes and flags can be attached to specific subsets of the data. A sort of audit trail is even provided, whereby datasets which are derived from other datasets can have their entire family trees laid out for the inspection of the users.

Also, and just as importantly, the format is flexible enough to be used for a wide variety of data fields and disciplines.

This document describes the df format, defining the structure of the bits which make up a dataset, explaining the concepts behind the format, giving examples of its use, and discussing implementation issues. While this introduction is intended for a wide audience, the rest of the document is heavily oriented towards computer programmers. (Physical scientists who are used to simply dumping their data arrays onto disk with no describing metadata should be forewarned: they will probably find the following chapters somewhat tedious and overcomplicated.)

Chapter 2

Specification of the Format

2.1 Definitions and Concepts

We define a *datum* to be a specific localized piece of information, consisting of one or more *components* which may be of any numeric or string type. Generally, both the datum and its individual components are identified with a quantity and physical units. Data are distributed in some sort of coordinate space indexed by one or more dimensional variables, each of which varies independently of the others. This implies the use of a general data array as our basic model of how data are accessed, with the proviso that irregularly distributed data must also be accommodated. The various dimensions which locate a datum, distinguishing it from other data, may then be identified tentatively with indices into an array. These indices, along with those which select components from a datum, tend to fall into a few functional groups, which we refer to as *dimensional levels*.

We designate indices which select components within a datum as *Level 0* dimensions. For example, a vector wind measurement might consist of the three spatial components of the wind (pointing in the direction of longitude, latitude, and pressure altitude); the index or variable which selects one component from these three would comprise the Level 0 dimension of the wind data.

The coordinates of a datum comprise the *Level 1* and *Level 2* dimensions of the data. Data are most likely to be stored in groups, or records. We therefore define the Level 1 dimensions as those coordinates which vary within a record; they are considered to vary solely within a group of data and are by nature fixed and bounded. Level 2 dimensions are those coordinates that vary across records. They are considered to vary across groups of data and may be unbounded.

Carrying the above example forward, each wind measurement will have a location in longitude, latitude, pressure, and time associated with it. Longitudes and latitudes might be considered fixed and bounded at, say, every 5 degrees; it seems natural to store these data as a series of two-dimensional arrays. The two dimensions, latitude and longitude, would then belong to Level 1. The other coordinates, pressure and time, would index these latitude-longitude arrays, and would therefore belong to Level 2.

Note that, if one chooses to store the data as three-dimensional arrays (latitude, longitude, and pressure), then those three dimensions would be Level 1, and only time would belong to Level 2. In other words, whether a dimension belongs to Level 1 or Level 2 is determined by how the data are stored, not by any intrinsic nature of the dimension. The nature of a dimension may, of course, influence how the data are written: time, being unbounded in most instances, will often be used as a Level 2 dimension.

This division into Level 1 and Level 2 dimensions also allows more flexibility in storing the data: As the Level 2 dimensions change, the range of the Level 1 dimensions may change as well. E.g., at different pressure levels, our latitude-longitude arrays of wind measurements may be of different sizes.

Finally, we also consider coordinates over which the data have been averaged in some way. While not used as indices into a data array, such dimensions are nonetheless extremely important in understanding the nature of the data; we refer to these as the *Level 3* dimensions. Continuing with our example, we may average the wind components over a month. We could indicate this averaging with Level 3 dimensions that identify each day of the month which contributed data to the average.

By *dimensional order*, we mean the number of dimensions at a dimension level. Using our (non-averaged) wind vector example above, the orders of the Level 0, Level 1, and Level 2 dimensions would be: three (for the East–West, North–South, and vertical components), two (for latitude and longitude), and two (for pressure and time), respectively.

The actual values along each dimension where the data are defined (not the data values themselves) we call the *grid points* of that dimension. For our wind vectors, the longitude dimension might be defined at every 5 degrees from 0 E to 355 E; there would be 72 grid points along this dimension. The latitudes vary from 90 S to 90 N by 5 degrees; this implies 37 grid points along the latitude dimension.

The *rank* of a dimension is the number of grid points along that dimension.

To fully describe the data with all its components and dimensions, we need other information which we refer to as *metadata* (data about the data). Metadata will identify the quantities and associated units that the data and dimensions represent. We also allow for the use of metadata in adding comment-type information to the data, as well as an audit trail; information on processing, packing and compression; and information about the physical form in which the data are stored.

In order to reduce the physical storage required by the metadata, to make it more machine-understandable, and to reduce the language dependencies that text comments impose, much of the metadata is stored in the form of integer codes. Some of the codes are centrally defined and uniform across all data sites; these codes are described in Section A.1. Other codes can be locally defined at a given site, and these are described in Section A.2. While the codes and their meanings are defined as part of this data format standard, we do not prescribe the form that the translation mechanism between these codes and their human-readable equivalents must take, although one current implementation based on using lookup tables is outlined in Appendix B.

Processing information is metadata which describes how the data were processed. For example, if a day of our wind data was missing, we might decide to interpolate in time to fill in that day's data. The processing information would consist of a code indicating time interpolation and providing the starting and ending days of the interpolation.

Auxiliary information is information which refers to or supplements the data. While processing information should be thought of as labels or tags applied to sections of the data, the auxiliary information on the other hand is more in the nature of a group of secondary variables which could not stand on their own as data in their own right, but which by their nature refer to the actual data. Examples would include instrument status words and uncertainties (error bars) in the data.

Packing information provides for the reduction of floating-point data to (usually smaller) integers by some scaling operation. This is different from compression, which uses an algorithm (such as Lempel-Zif) to reduce the data to a smaller, encoded sequence of bytes. Data can be both packed and compressed. For example, we may take our wind measurements and scale them at each pressure level to more compact integers. We could then use a compression routine to compress this packed data into a byte array whose length is determined by the bit patterns in the data. To read the data, we would first uncompress it and then unpack it using the metadata provided in the specification of the winds.

The collection of the data, the specification of its dimensions, and its metadata are collectively referred to as a *data object*.

Finally, at the highest level of organization, we define a *dataset* as a collection of data objects. Note that we avoid the use of the term "file," since we wish the proposed standard to be applicable to data storage and access entities other than simple files on disk.

2.2 Overview of the Dataset Structure

A dataset written in the df standard consists of a series of records. A record is defined to be a series of bytes which are to be read or written together. An example would be a Fortran record as written with a single WRITE statement. The concept of a record is especially important for certain languages (such as Fortran) and for certain binary data representations (such as XDR). On the other hand, a record might consist of a set of bytes in a uniform bit stream with no delimiters. Each record defined in the df format begins with a specific numeric code which identifies its type. Record types are organized into functional groups. Within a group, and from group to group, a lower numeric ID code generally indicates its placement further towards the beginning of the dataset.

Because the df format should be usable with any storage architecture or language, it was designed to function with purely sequential data streams used with languages (such as Fortran) which need to know how long a record is before it is read. These design constraints do not prevent datasets from being accessed in a non-sequential fashion or by more sophisticated languages, but the format must at least work for this lowest common denominator.

In order that the next record type in a dataset be known (so that its length is known as well), the order of the records is fixed. In addition, the length of some records depends upon information read in previous records.

All of the information needed to read and decode the dataset is present in the dataset. Metadata is recorded in appropriate records in the proper record groups. Some groups are composed of only one record type, while others consist of several. Some record types are mandatory, and others are optional. Some types of records can be written more than once, and these will be further indexed so that they can be processed in a particular order. See Section 2.3.2 for a discussion of what indices mean in various contexts.

Following the metadata come the data records themselves. The information in the previous records fully describe the structure and format of the data, making it possible to read.

2.2.1 Dataset Structure

The dataset structure may be specified in Backus-Naur Form as:

```
<Dataset> ::= <TEST> <Object> { <Object> }

<Object>  ::= <OBJDESC>  <AUDIT>  [ <INFOSPEC> ]
              { <COMMENT> } <DIMSPEC0> <DIMSPEC1>  <DIMSPEC2>
              [ <DIMSPEC3> ] <DESCRIP0> <Dimgroup1>
              { <Dimgroup1> } <Dimgroup2> { <Dimgroup2> }
              [ <Dimgroup3> { <Dimgroup3> } ]
              { <BADVAL> } { <Procgroup> }  { <Auxgroup> }
              <Regdata> | <Packdata> | <Compdata>

<Dimgroup1> ::=  <DESCRIP1>  <DESCVAL>  [ <DESCSUP> ]
                 { <Descgroup> }

<Dimgroup2> ::=  <DESCRIP2>  <DESCVAL>  [ <DESCSUP> ]
                 { <Descgroup> }

<Dimgroup3> ::=  <DESCRIP3>  <DESCVAL>  [ <DESCSUP> ]
                 { <Descgroup> }

<Descgroup> ::=  <DESCRIP>  <DESCVAL>  [ <DESCSUP> ]

<Procgroup> ::=  <PROCSPEC>  [ <PROCFORM>  <PROCVAL> ]
                 { <PROCDUP> }

<Auxgroup> ::= <AUXSPEC> <AUXRANGE> <AUXVAL> [ <AUXSUP> ]

<Regdata> ::= <REGDAT> { <REGDAT> }

<Packdata> ::= <Packgroup> { <Packgroup> } <PAKDAT>
               { <PAKDAT> }
```

```
<Compdata> ::= <Compgroup> <COMPDAT> { <COMPDAT> }

<Packgroup> ::=  <PAKSPEC>  [ <PAKFORM>  <PAKVAL> ]

<Compgroup> ::=   <COMPSPEC>  <COMPLEN>
                  [ <COMPFORM>  <COMPVAL> ]
```

where `<UPPERCASE>` indicates a record type, `<Mixedcase>` represents a group of records, braces ("{ }") indicate zero or more occurrences of what they enclose, brackets ("[ ]") indicate zero or one occurrences of what they enclose, and a vertical bar ("|") indicates a mutually exclusive choice among options.

2.2.2 Record Type Descriptions

A brief overview of each of the record types follows. See Section 2.4 for the details of each record type.

TEST provides all of the system-dependent information which is necessary for interpreting the bit patterns in the subsequent dataset records. It also contains a magic number which identifies that the dataset is in the df format.

OBJDESC describes a data object. It contains general information needed by the other record types.

AUDIT provides for an audit trace of data objects which have been derived from other data objects. See Section 2.3.4 for a discussion and examples of the audit tree.

INFOSPEC is a record type that contains unrestricted user-specified information. See Section A.2.6 for details.

COMMENT contains user-specified information in the form of an 80-character string. This allows for human-readable comments.

DIMSPEC0 specifies how the Level 0 dimensions of the data object are indexed.

DIMSPEC1 specifies how the Level 1 dimensions of the data object are indexed.

DIMSPEC2 specifies how the Level 2 dimensions of the data object are indexed.

DIMSPEC3 specifies the number of DESCRIP3 records for the Level 3 dimensions of the data object.

DESCRIP0 describes the data or its component parts that each Level 0 dimension represents.

DESCRIP1 describes each Level 1 dimension. This is the authoritative descriptor for a Level 1 dimension.

DESCRIP2 describes each Level 2 dimension. This is the authoritative descriptor for a Level 2 dimension.

DESCRIP3 describes each Level 3 dimension. This is the authoritative descriptor for a Level 3 dimension.

DESCRIP provides an alternative, nonauthoritative description of any Level 1, Level 2, or Level 3 dimension. These descriptors are a means of "renaming" grid points in terms which may be more meaningful to the dataset's users.

DESCVAL contains the grid point values corresponding to a dimension described by a DESCRIP1, DESCRIP2, DESCRIP3, or DESCRIP record type.

DESCSUP contains supplementary information about a given dimension. For example, if a dimension corresponds to an x or y coordinate in a geographical map projection, then the coordinates of the map pole would be specified here.

BADVAL specifies how missing or bad data are to be flagged in the data records. The absence of BADVAL records implies that there are no bad or missing data.

PROCSPEC provides a means for attaching special notes or processing flags to subsets of the data. See Section 2.3.5 for more details.

PROCFORM specifies the format of the processing information in the associated PROCVAL record. If the processing method used requires no information values, then this record will not be present.

PROCVAL specifies the processing information as indicated in the associated PROCSPEC record. If the processing method used requires no information values, then this record will not be present.

PROCDUP allows for processing information specified by a previous Procgroup to be applied over a different range of dimensions. This record provides a shorter alternative to duplicating an entire Procgroup with a different set of coordinate values.

AUXSPEC specifies the nature of the auxiliary information and how it applies to the data. See Section 2.3.3 for more details.

AUXRANGE specifies the dimensions the auxiliary information refers to and which dimensions it varies over.

AUXVAL gives the actual auxiliary information itself.

AUXSUP gives certain supplemental values which may be needed to interpret the auxiliary values.

PAKSPEC describes the method used to pack the data in the data object.

PAKFORM specifies the format of the packing information in the associated PAKVAL record. If the packing method used requires no parameters, and no bad data are present, then this record will not be present.

PAKVAL specifies the packing information as indicated in the PAKSPEC record. If the packing method used requires no parameters, and no bad data are present, then this record will not be present.

COMPSPEC describes the compression method used to compress the data in the data object.

COMPLEN gives the length of each COMPDAT record of compressed data.

COMPFORM specifies the format of the compression information in the COMPVAL record. If the compression algorithm requires no parameters, then this record will not be present.

COMPVAL specifies the compression information as indicated in the COMPSPEC record. If the compression algorithm requires no parameters, then this record will not be present.

REGDAT contains the actual data values, if the data have not been packed or compressed. Each record contains the data over a single set of values of the Level 2 dimensions: Level 1 dimensions vary *within* a REGDAT record, while Level 2 dimensions vary *between* REGDAT records.

PAKDAT contains the actual packed data values, if the data have been packed. PAKDAT has the same structure (number of records, number of data values in each record) as a REGDAT record, but it contains packed data.

COMPDAT contains the actual compressed data values, if the data have been compressed. Usually there will be only one COMPDAT record.

2.3 Elucidation

2.3.1 How dimensions are specified

We consider the dimensions of a data object to be those quantities which are used to select and categorize the data values. To understand the nature of these dimensions, it may help to identify them with the indices of a multi-dimensional array. (This paradigm has its limits, though: data which cannot be described in terms of a regular array can nevertheless be described in terms of these dimensions.) We have proposed four levels of these dimensions:

Level 0 dimensions describe the components of the data (these dimensions are unrelated to the position of a datum in a coordinate space). Simple scalar data such as temperature would have a single Level 0 dimension containing a single grid point (i.e, A Level 0 of order 1 with a rank of 1). A two-component horizontal wind vector would also have a single Level 0 dimension, but with two grid points (order of 1, rank of 2). A wind stress tensor (a 3×3 matrix) would have two Level 0 dimensions, each of which would have three grid points (order of 2, ranks of 3 and 3). In terms of array indices, Level 0 dimensions select components at a fixed position. That is, a wind stress tensor is considered to be a datum, and two Level 0 indices are needed to select a single component of the tensor. Likewise, a wind vector as a whole is a single item, but a Level 0 index would select which component of the wind (North–South or East–West) is desired. For temperature, there is only one component to select; this would correspond to a single array index which can take only a single value. (Because a $N \times M$ array is indistinguishable from a $N \times M \times 1$ array, such an index is superfluous and can be omitted.)

Level 1 dimensions are those that occur within a single data record.

Level 2 dimensions are those that occur across data records. The Level 1 dimensions, together with the Level 2 dimensions, locate a datum in a coordinate space. The difference between the two levels is that Level 1 dimensions vary within a data record in the dataset, while Level 2 dimensions vary between data records. If each data record were read into a separate array variable, then every array would have to have an index for each Level 0 and Level 1 dimension. Suppose, for example, that a set of temperatures is written out in a series of two-dimensional longitude-latitude grids, one for each day. The resulting dataset would have two Level 1 dimensions—longitude and latitude—plus one Level 2 dimension: time.

This distinction between the two levels may seem artificial, but it has two advantages: first, it enables a program to read the data either as a single large array variable (whose indices would correspond to each of the Level 0, Level 1, and Level 2 dimensions), or as a series of separate variables (whose array indices correspond to each of the Level 0 and Level 1 dimensions, with the number of arrays calculated from the Level 2 dimensions). Second, if the data are read in as separate arrays, then those arrays may be of different sizes. That is, the Level 1 dimensions may have different structures over different ranges of Level 2 dimensions. For example, the first five days of the longitude-latitude wind fields might consist of five $2 \times 72 \times 37$ arrays, while the next ten days might be a series of ten $2 \times 144 \times 46$ arrays.

Level 3 dimensions specify how the data in the dataset have been averaged, integrated, or summed over which subsets of Level 0, Level 1, and Level 2 dimensions. These are "virtual dimensions", in that they do not correspond to any indices in a data array, but otherwise their structure is very similar to that of the other, "real" dimensions. A set of monthly averaged data, for example, would have a Level 3 dimension corresponding to time and detailing the days of the month over which the average was taken, as well as what averaging method was used. (Note that instantaneous or point data, consisting of observations or calculations that are considered to occur at fixed points in coordinate space, have no averages and hence have no Level 3 dimensions.)

Dimensions may have multiple, parallel definitions. For example, pressure levels at which data are recorded can also be specified as altitudes above sea level. Which description is best? This is a decision best left to the dataset originator. The originator must define one quantity as the authoritative descriptor for a given dimension; any other descriptors for that same dimension will be duplicate, additional descriptors for the dimension. A user of the data will use the authoritative descriptor to obtain the definitive word on the grid point values of that dimension; users may use the additional information if they desire it, but that information is not required for understanding the quantity that dimension represents.

2.3.2 Indices

The df format uses several kinds of indices to provide flexibility in how a dataset can be written. Indices are used in three basic ways: to identify or order the type of each record, to order the information about the data dimensions, and to specify ranges over which various conditions apply.

Examples of the use these various indices may be found in Chapter 3.

Record indices

Each record begins with an integer code indicating what kind of record it is. This record ID can also be used as an ordering index. Currently, the order of records in a dataset is fixed, and this ordering function is somewhat redundant.

Some record types and record groups can be repeated (i.e., DESCRIP1, DESCRIP3, COMMENT, BADVAL, Procgroup, Auxgroup, and Packgroup); these records all include a RECSORT field which is used to associate related records within a group and to order different records (or record groups) which are of the same type. Except for BADVAL and DIMSPEC1 records (whose multiple records are not allowed to overlap), this ordering may be very important: where conflicts arise in regions where two records' domains overlap, the record with the higher-numbered index takes precedence.

For example, the COMMENT records are ordered by their indices such that, even if they are written to the dataset in a random order, a reader program can know how to put them back in order, so that the sentences, tabular values, or other such information in the comments makes sense.

The RECSORT fields in the Procgroup and Packgroup records specify the order in which subsets of the data were processed or packed. (These subsets are specified by other fields, START and END, which are described in Section 2.3.2). The Auxgroup RECSORT fields are mainly a convenience to distinguish one record from another within an Auxgroup. The information contained in the Procgroup and Auxgroup records is usually not needed to read the data and could be ignored; this information is often, however, needed to use the data *wisely*. The Packgroup records, on the other hand, contain information which is critical to reading the data, and their ordering must be specified with care.

The DESCRIP1 and DESCRIP3 records use the RECSORT field to distinguish between descriptors of different subsets the same dimension index. The order of the DESCRIP1 records is not important since there can be no overlap in specification. The order of the DESCRIP3 records is important and indicates the order in which the data were averaged.

Dimension Indices

Returning to the identification of the various dimensional levels with sets of indices of a data array, one realizes that the description given in Section 2.3.1 implies a certain ordering to that array: Level 0 dimensions/indices would be the fastest-varying, followed by the Level 1 dimensions/indices, and finally the Level 2 dimensions/indices. To impose such an ordering on how the data sets are written would be excessively restrictive, so a logical separation is made between the order of the array indices (which may be in any arbitrary order desired by the dataset creator) and the order in which dimensions are specified, which is fixed by the format. A pointer or index must therefore be attached to each dimensional specification, indicating to which "array index" that dimension corresponds. The set of all these pointers, then, would constitute a one-to-one mapping between array indices and data dimensions. (A special case of such a pointer is the Level 0 dimension of scalar data; as mentioned above, a dimension of rank 1 is superfluous, and, since there is therefore unlikely to be an array dimension corresponding to the single Level 0 grid point, the pointer may be taken as -1—it doesn't really point to an array index.).

For convenience, we will refer to the data array as written in the dataset as the "actual array," while the data array whose indices are ordered in the Level 0–Level 1–Level 2 sequence will be called the "virtual array." In this virtual array, the first I indices correspond to the Level 0 dimensions (where I is the order of the Level 0 dimensions—the number of Level 0 dimensions), the next J indices correspond to the Level 1 dimensions (J being the order of the Level 1 dimensions), and the final K indices correspond to the Level 2 dimensions (where K is the order of the Level 2 dimensions).

The pointers which map the virtual-array indices to the actual-array indices are specified in the INDEX fields of the DIMSPEC0, DIMSPEC1, and DIMSPEC2 records. Each INDEX field is a one-dimensional array of pointers; the $i\mathrm{th}$ element of an INDEX field gives the position in the actual array of the index of a given Level's $i\mathrm{th}$ dimension. For examples, see Section 3.2.

The DESCRIP1, DESCRIP2, DESCRIP3, DESCRIP, DESCVAL, and DESCSUP records contain another index, NDEX, which identifies which dimension at a given level they are describing. This NDEX value reflects the ordering of the indices of the virtual array, however, and not the actual array's indices. That is, the field DESCRIP1.DEXSORT.NDEX is to be used as an index into the DIMSPEC1.INDEX pointer array: DIMSPEC1.INDEX[DESCRIP1.DEXSORT.NDEX] gives that index of the actual array which corresponds to the dimension being described by the DESCRIP1 record. (This sounds more complicated than it actually is; see the examples in Section 3.2.)

START and END Fields

START and END are index fields contained in the DESCRIP1, DESCRIP3, BADVAL, PROCSPEC, PROCDUP, AUXSPEC, and PAKSPEC record types. These records can apply over specific, limited ranges of the data dimensions. For example, a BADVAL record can define a bad-data flag value which applies only to a specific subset of the data: any values found in that subset which match the bad-data flag are considered to be bad or missing data, while the same values found outside the subset are treated as normal, legitimate data values.

The START and END indices are the mechanism for defining such subsets. Both START and END are one-dimensional arrays (vectors) of indices. These are indices into the *virtual* array defined in Section 2.3.2. If we denote the order of the Level 0, Level 1, and Level 2 dimensions by I, J, and K, respectively, then a PROCSPEC.START array is $I + J + K$ elements long. If a record type is to apply over a specific range of values in each dimension, the START indices specify the beginnings of those ranges, and the END indices specify their ends. (Note that an index of -1 represents the last element of a dimension.)

Some START and END records do not cover all the indices of the virtual array. Every BADVAL record, for example, must specify bad-data flags for all grid points in all Level 0 dimensions; its START and END fields refer only to ranges in the Level 1 and Level 2 dimensions and are hence $J + K$ elements long.

Likewise, DESCRIP1, which must specify which data records are described by the Level 1 dimension being defined, can cover ranges only of the Level 2 dimensions. Its START and END fields, then, are only K elements long.

For examples, see Section 3.3.

Auxiliary information array indices

See Section 2.3.3 for a description of how the auxiliary information arrays are arranged.

2.3.3 Auxiliary Information

In discussing auxiliary information, it may be useful to keep in mind a particular example of such information: the uncertainty in a measurement. Uncertainty generally does not stand alone on its own merits as a variable in its own right, so it does not belong in the data. On the other hand, it is too important to be left to local processing codes (not to mention that fact that, if it varies enough over the data, it could use up a considerable amount of disk space if put into the processing codes, since for every PROCVAL there must be a PROCFORM.) Hence, the existence of a separate group of metadata: auxiliary information.

Note that the auxiliary information can refer not only to the data, but to the dimensions as well. For example, temperatures (data) may have uncertainties attached, but so can the altitudes (dimension) at which those temperatures were taken. Therefore, the auxiliary information *refers* to one set of dimensions (including Level 0 dimensions if it refers to the data), and it will at the same time *apply* to another set of dimensions–that is, it may vary over a range of those other dimensions.

If the uncertainties in the temperatures in our example vary from day to day, then we must specify not only the dimensions to which the auxiliary information refers, but also the dimension (time) over which it varies, and we must include in the auxiliary values a set of uncertainties applicable for each day. The auxiliary information, then, is said to apply over the time dimension.

These two concepts (reference and application) can be summarized by stating that the reference dimensions tell what the auxiliary information is about; the application dimensions tell how it varies over the data field.

The quantity code associated with auxiliary information may be a quantity such as "error" or "uncertainty," in which case the quantity or quantities of the data or dimensions to which it refers are the actual physical quantities involved. In addition, auxiliary information may carry its own units. If it is marked as having no units, either there are no units for the auxiliary information or the units are to be taken from the variables being referred to.

The auxiliary information itself is contained in an array, and some correspondence must then be established between the elements of that array and the data and its dimensions. To do this, two fields are defined: APPLEV and APPNDEX, lists of dimensional levels and dimensional indices, respectively, whose elements correspond to the associated dimensions of the auxiliary array.

Thus, the number of application dimensions determines the number of dimensions of the auxiliary information array. The size of the auxiliary array (i.e., the number of grid points in each of its dimensions) is determined by the START–END indices associated with the Auxgroup. Thus, to determine the size of the i'th dimension of the auxiliary array, one would look at the i'th element of the application dimension list, obtain the level and index of the data dimension to which the array dimension corresponds, and use that level and index to look into the START and END lists for the beginning and ending data dimension grid points over which the Auxgroup applies. The number of grid points is then the same as the size of the auxiliary array along that array dimension.

A similar mechanism exists to indicate which data dimensions the Auxgroup refers to. Again, two fields are defined: REFLEV and REFNDEX, which contain the levels and indices of the data dimensions referred to by the auxiliary information. The order of the elements of these two lists is unimportant. The extent of a dimension over which the Auxgroup refers is specified by the Auxgroup's START and END fields. A reference dimension may also appear in the application dimensions' list; this indicates that the auxiliary information not only refers to that dimension, but applies to it as well (i.e., varies over its grid points as well). If a reference dimension is *not* repeated in the application dimensions' list, then all of the auxiliary information refers to each grid point in that reference dimension (as bounded by its START and END points).

2.3.4 Audit Trail

The audit trail concept is a very important one in trying to trace the history of a dataset and can be especially useful when one is faced with an unknown dataset. It allows a user to discover the family history of the data, finding out what datasets were used in creating the present data, from which sites these ancestor datasets originated, and which tasks at those sites produced them.

Such a family tree is passed down from dataset to dataset, generation to generation, by extracting it from component datasets, joining it together somehow, and putting the result into a new dataset. Because data from any dataset may be used to create some other dataset (e.g., to initialize a model), such an audit trail must be included in *every* dataset if it is to be useful. Care must be taken, therefore, when designing such a data structure to keep it compact, including only the information which is absolutely necessary. Including all relevant information (file names, program names and version numbers, run-time parameters) would cost more in increased file sizes than would be gained in usefulness. Therefore, the df audit trail is made up of nodes, one per ancestor dataset, which are composed of four long integers: a site ID, a task ID, a date stamp, and a pointer to another audit node. This information is limited, but it does allow a user to contact the sites for further information beyond what is given in the dataset's COMMENT, INFOSPEC, and PROCSPEC records.

The AUDIT record type keeps track of this audit trail with its TREE field, which contains a group of nodes connected in a tree structure. The site identifier code in a node is unique to the site where the df format dataset is generated. (These codes are discussed in Section A.1.2). The task identifier is a site-dependent code indicating the task which generated the data. (These codes are explained in Section A.2.1). The date stamp is the the day number from 1 January 1900 (i.e., the number of days elapsed since 31 December 1899). The pointer points to the next node in the tree (with the the current data set's pointer being NULL. The tree is set up as a one-dimensional array of nodes. Each pointer contains a displacement from itself in the array: for example, a pointer to the next node would have a value of 1, a pointer the the third node down the line would have a value of 3, and so on.

See Section 3.4 for examples.

2.3.5 Processing Codes

Processing codes are intended as notes which are appended to subsets of the data. Suppose, for example, that an array of data has a gap of bad or missing data, and that the most frequent use of the data requires that the gap be filled by some form of interpolation. For efficiency, it is desirable to write the interpolated data into the file with the real data, so that the interpolation will not have to be done again and again, each time the dataset is read. On the other hand, it would be very dangerous to simply mix the interpolated and real data in the dataset indiscriminately.

Processing codes are meant to handle such situations. One can use Procgroup record groups to mark any part of the data to which the attention of the cautious user needs to be drawn.

2.4 Record Type Format Specifications

This section contains the specifications for each record type. Each field will be described and the format specified by one of the following codes:

B	=	byte
C	=	character
F	=	single precision floating-point
I	=	long integer
N	=	audit node (I, I, I, I)
S	=	string
?	=	type as specified by a field in an earlier record
*	=	array

2.4.1 TEST

This must be the first record in any dataset. This contains all of the information needed to determine how to read the dataset. The information is stored in 8-bit bytes so that it is machine-independent. A reader program on any computer system should be able to determine the format of the dataset, whether or not it can read that dataset, and how to read it.

The dataset is defined to begin at the TEST record, no matter what information appears before it. It is permissible to include miscellaneous information before this record in the dataset (such as a Standard Format Data Unit label); such prepended information will not be considered to be a part of the dataset for the purposes of this standard. If the dataset is written with some sort of header bytes before the TEST record (e.g., the byte array header in the XDR format) then the dataset is defined to start with that header. The presence of these headers can be deduced from the contents of the TEST record. Thus, when trying to identify a dataset of unknown type, a program should scan until it encounters the magic number, read the next 20 bytes, and determine whether header bytes are present

The record consists of a total of 24 bytes.

name	bytes	description
MAGIC	1-4	contains the magic number: 47 f3 46 e3 in hexadecimal notation. The bytes must be specified in exactly this order. The dataset can be searched byte for byte until this sequence is found. The odds of this sequence occurring at random is 1 in 4294967296. Therefore, there is a slight chance of a false identification of a dataset being in the standard format. There is also a very slight chance that prepended information in a standard dataset will be identified as the start of the dataset.
MACHID	5	contains an identification number of the machine that generated the dataset. This field is for tracking the history of the dataset and is not needed for interpreting the dataset. The codes are as follows:

- 0 = unknown (free use of this code is discouraged)
- 1 = DEC VAX running VMS
- 2 = Silicon Graphics Iris workstation
- 3 = Cray Y-MP running UNICOS
- 4 = IBM mainframe running MVS
- 5 = MS-DOS personal computer
- 6 = Apple Macintosh
- 7 = Sun workstation
- 8 = DEC Ultrix workstation
- 9 = Hewlett-Packard Unix workstation
- 10 = IBM Unix workstation
- 11 = Convex

NUMOBJECTS 6 is the number of data objects contained in the dataset. If zero, then there is assumed to be one data object

SPECA 7 contains information about the byte and word formats of the data. Bits are specified as follows (counting from the least significant bit):

CHARSET (bits 0-1) indicates which character set is used for text as follows:
- 0 = XDR (subsequent bits in this byte are ignored)
- 1 = ASCII
- 2 = EBCDIC

BSWAP (bit 2) indicates what kind of byte-swapping is used as follows:
- 0 = most significant byte is stored in the lower of the two machine word addresses
- 1 = most significant byte is stored in the higher of the two machine word addresses

WSWAP (bit 3) indicates what kind of word-swapping is used as follows:
- 0 = most significant word is stored in the lower of the two machine word addresses
- 1 = most significant word is stored in the higher of the two machine word addresses

RESERVE (bits 4-7) is reserved for future expansion

RESERVED1 8 is reserved for future expansion

RECHDR	9	indicates what kind of operating system and language record headers are present. The intent here is to specify which algorithm to use in tracing through any headers present, not to specify which machine wrote the dataset. This field is completely independent of the MACHID field. A machine may be able to read and write headers from foreign machines. The codes are specified as follows: 0 = XDR 1 = none (pure stream file) 2 = VMS Fortran segmented variable length 3 = f77 Fortran 4 = Cray COS 5 = IBM VBS
RESERVED2	10	is reserved for future expansion
SPECB	11	contains information about array ordering and index references. Bits are specified as follows (counting from the least significant bit): **ARRORD** (bit 0) indicates the ordering of elements in multidimensional arrays: 0 = fastest-varying index is last (most languages) 1 = slowest-varying index is last (Fortran, IDL) **IDXSTART** (bit 1) indicates the starting value of indices: 0 = start from 0 1 = start from 1 **RESERVE** (bits 2-7) is reserved for future expansion
SLEN	12	indicates the length in bits of a short integer 0 = XDR
LLEN	13	indicates the length in bits of a long integer 0 = XDR
FLEN	14	indicates the length in bits of a single precision floating-point number 0 = XDR
DLEN	15	indicates the length in bits of a double precision floating-point number 0 = XDR
FPFORM	16	contains information about the form of floating-point numbers. Bits are specified as follows (counting from the least significant bit):

		SP (bits 0-3) indicates the format code of a single precision floating-point number: 0 = XDR 1 = IEEE 2 = VAX 3 = IBM mainframe 4 = Cray **DP** (bits 4-7) indicates the format code of a double precision floating-point number: 0 = XDR 1 = IEEE 2 = VAX D 3 = IBM mainframe 4 = Cray 5 = VAX G
RESERVED3	17	is reserved for future expansion
RESERVED4	18	is reserved for future expansion
RESERVED5	19	is reserved for future expansion
RESERVED6	20	is reserved for future expansion
RESERVED7	21	is reserved for future expansion
RESERVED8	22	is reserved for future expansion
RESERVED9	23	is reserved for future expansion
RESERVED10	24	is reserved for future expansion

2.4.2 OBJDESC

This record type begins the specification of each data object. It contains information on the enumeration of dimension orders and the number of items for various descriptor record types.

name	format	description
RECTYPE	I	is the record type code = 1
VARTYPE	I	is the physical quantity code for the data object. This code is explained in Section A.1.3
RESERV0	I	is reserved for future expansion
NDIM0	I	is the number of Level 0 dimensions (Level 0 order)
NDIM1	I	is the number of Level 1 dimensions (Level 1 order)
NDIM2	I	is the number of Level 2 dimensions (Level 2 order)
NDIM3	I	is the number of Level 3 dimensions (Level 3 order)
RESERV1	I	is reserved for future expansion
ISOURCE	I	is the data source code. This code is explained in Section A.2.2
RESERV2	I	is reserved for future expansion
RESERV3	I	is reserved for future expansion
NAUDIT	I	is the number of nodes in the AUDIT.TREE field. (Must be greater than zero)
NINFO	I	is the number of bytes in the INFOSPEC.INFO field. If zero, then there will be no INFOSPEC record
NCOMMS	I	is the number of COMMENT records which follows. If zero, then there will be no COMMENT records
COMCOD	I	is the character string format code for the COMMENT.NOTES field. This code is explained in Section A.1.1
RESERV4	I	is reserved for future expansion
NBADS	I	is the number of BADVAL records which follow. If zero, then there are no BADVAL records, which implies that there are no bad or missing data in this data object
RESERV5	I	is reserved for future expansion
NPROCS	I	is the number of Procgroups which follow. If zero, then there are no Procgroup records, which implies that there is no processing information
RESERV6	I	is reserved for future expansion
NPACKS	I	is the number of Packgroups which follow. If zero, then there are no Packgroup records, which implies that the data are not packed
RESERV7	I	is reserved for future expansion
NAUX	I	is the number of AUXSPEC records which follow. If zero, then there will be no AUXSPEC records.

CMPNUM	I	is the number of COMPDAT records present in the dataset, as well as the number of elements in the COMPLEN.LENGTHS record field which follows. Normally, there will be only one COMPDAT record (or none at all), but for especially long files, or given constraints of certain computer languages, it may be advisable to split the compressed data into several records of arbitrary lengths, recorded here. If CMPNUM is zero, then there are no Compgroup or COMPDAT records
RESERV8	I	is reserved for future expansion
RESERV9	I	is reserved for future expansion
RESERV10	I	is reserved for future expansion

2.4.3 AUDIT

This record type specifies an audit trace of how the data was processed. This “family tree” of the data is implemented as a vector of nodes, each of which represents a step in the history of the data. Each node refers to a dataset used in the creation of the current data object.

name	format	description
RECTYPE	I	is the record type code = 10
TREE	N*	contains the tree structure specifying the audit trail of the data object. Each node in the tree contains the four parts of the audit structure: the site identifier (Site IDs are centrally registered and are discussed in A.1.2,) the task identifier (Task codes are site-defined and are discussed in A.2.1,) the date of generation, and a relative pointer to the next lower node in the tree. The last (root) node must contain the identifiers and time stamp for the current data object and a NULL pointer. See Section 2.3.4 for details on this structure

2.4.4 INFOSPEC

This record type allows for the specification of general information that is completely user-defined. The df standard provides no general mechanism for interpretation of these bytes. The number of bytes in the INFO field are specified by OBJDESC.NINFO. If OBJDESC.NINFO = 0, then this record will not be present.

name	format	description
RECTYPE	I	is the record type code = 11
INFO	B*	contains the information as an array of bytes. The user is encouraged to document the meaning of these bytes in the COMMENT field or in a standard site-dependent file. See the discussion in Section A.2.6

2.4.5 COMMENT

This record type allows for the specification of general information that is completely user-defined. These records are in human-readable format. The number of records is specified in OBJDESC.NCOMMS. The records are considered to be in the order specified in RECSORT, regardless of their actual physical order in the dataset.

name	format	description
RECTYPE	I	is the record type code = 12
RECSORT	I	is the index for sorting the records. Lower indexed records will be processed before higher indexed records. See the discussion in Section 2.3.2
NOTES	S	is an 80-character string containing human-readable text. The string is formatted as in the OBJDESC.COMCOD field

2.4.6 DIMSPEC0

This record type specifies the Level 0 dimensions (components) of the data object. The field arrays are OBJDESC.NDIM0 elements in length.

name	format	description
RECTYPE	I	is the record type code = 20
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
INDEX	I*	is the relative index pointer for the Level 0 dimensions. These indices are pointers that map the ordering of the DESCRIP0 fields into the actual data array index order. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. A value of -1 indicates a scalar value (i.e., there is no separate index value into the data array). See the discussion in Section 2.3.2
GPTNUM	I*	is the number of grid points defined along each dimension (the rank of the dimension). These values must be greater than zero

2.4.7 DIMSPEC1

This record type specifies the Level 1 dimensions of the data object. There can be several DESCRIP1 records for each Level 1 dimension. The field arrays are OBJDESC.NDIM1 elements in length.

name	format	description
RECTYPE	I	is the record type code = 21
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
INDEX	I*	is the relative index pointer for the Level 1 dimensions. The values in this field are pointers that map the ordering of the DESCRIP1 records into the actual data array index order. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
DESNUM	I*	is the number of DESCRIP1 records used to describe this dimension. The user can specify a different number of Level 1 dimension grid points along various Level 2 dimensions; these different Level 1 dimension definitions require different DESCRIP1 records, whose number is given here. Note, however, these differing Level 1 dimensions must be defined over non-overlapping ranges of Level 2 dimensions. DESNUM values must be greater than zero

2.4.8 DIMSPEC2

This record type specifies the Level 2 dimensions of the data object. There must be one DESCRIP2 record for each Level 2 dimension. The field arrays are OBJDESC.NDIM2 elements in length.

name	format	description
RECTYPE	I	is the record type code = 22
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
INDEX	I*	is the relative index pointer for the Level 2 dimensions. The values in this field are pointers that map the ordering of the DESCRIP2 records into the actual data array index order. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
GPTNUM	I*	is the number of grid points defined along each dimension (the rank of the dimension). These values must be either greater than zero or equal to -1 (the latter to indicate unboundedness)

2.4.9 DIMSPEC3

This record type specifies the Level 3 (averaging) dimensions of the data object. Several DESCRIP3 records can exist for every Level 3 dimension. The field array is OBJDESC.NDIM3 elements in length.

name	format	description
RECTYPE	I	is the record type code = 23
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
DESNUM	I*	is the number of DESCRIP3 records used to describe this dimension. These values must be greater than zero

2.4.10 DESCRIP0

This record type contains the descriptions of the Level 0 dimensions. The length of each array is the multiplicative sum of the elements in DIMSPEC0.GPTNUM.

name	format	description
RECTYPE	I	is the record type code = 30
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
DATFMT	I*	is the data format code. If the data are packed, this will be the format of the data after they have been unpacked. This code is explained in Section A.1.1
VARTYPE	I*	is the quantity code for each data component (Level 0 dimension). This code is explained in Section A.1.3
UNITS	I*	is the units code for each data component (Level 0 dimension). This code is explained in Section A.1.4

2.4.11 DESCRIP1

For each Level 1 dimension (i =1,...,OBJDESC.NDIM1) there must be DIMSPEC1.DESNUM[i] records in this record type. The descriptor records may be in any order, but it is recommended that they be in order of 1 through OBJDESC.NDIM1. This record type is considered the authoritative source of information for this dimension. A parallel description can be placed in the DESCRIP record type. There must be a DESCVAL record for each DESCRIP1 record, to specify the grid points along that dimension. An optional DESCSUP record may be present if additional information is required. An important note: There should be no overlap in the ranges of the Level 2 dimensions, since there cannot be two differing number of grid points along any particular Level 2 dimension. The START and END fields define the ranges of the Level 2 dimensions to which these particular Level 1 dimensions apply. For example, the user may have observations at particular spatial locations that have been made at varying height levels. The spatial locations might make up the Level 2 dimension and the height levels would serve as a the Level 1 dimension. The START and END fields will contain the particular spatial locations that a particular set of height levels apply to.

name	format	description
RECTYPE	I	is the record type code = 31
DEXSORT	I	**NDEX** (bits 0-15) specifies which Level 1 dimension is being described. Counting begins at 0 or 1 as specified in the TEST.SPECB.IDXSTART field. Note that this index does not refer to an index position of an actual data array but is instead a pointer into the DIMSPEC1.INDEX field. See the discussion in Section 2.3.2 **RECSORT** (bits 16-31) is the index for sorting DESCRIP1 records with the same NDEX field. See the discussion in Section 2.3.2
START	I*	is an OBJDESC.NDIM2-element array of Level 2 dimension grid points over which this record begins to apply. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an OBJDESC.NDIM2-element array of Level 2 dimension grid points over which this record stops applying. A value of -1 indicates the last grid point of that dimension. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
GPTNUM	I	is the number of grid points along this dimension
DUPNUM	I	is the number of DESCRIP records for this dimension
DESSUP	I	is the number of floating-point values in the SVALS field of the following DESCSUP record. If zero, then there is no DESCSUP record for this dimension
DESFMT	I	is the descriptor format code. This code is explained in Section A.1.1

DESTYPE	I	is the quantity code for the physical quantity that the dimension represents. This code is explained in Section A.1.3.
UNITS	I	is the units code for the physical quantity that the dimension represents. This code is explained in Section A.1.4.
STORG	I	specifies how the values of the descriptor are stored in the DESCVAL record which follows. The possible values are: 0 = an explicit array of GPTNUM values 1 = an implicit specification consisting of a (low-value, interval) pair 2 = an implicit specification consisting of a (low-value, high-value) pair
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.12 DESCRIP2

For each Level 2 dimension (i =1,. . . ,OBJDESC.NDIM2) there must be a single record in this record type. The descriptor records may be in any order, but it is recommended that they be in the order 1 through OBJDESC.NDIM2. This record is considered the authoritative source of information for this dimension. A parallel description can be supplied in the DESCRIP record type. There must be a DESCVAL record for each DESCRIP2 record which specifies the grid points along that dimension. An optional DESCSUP record may be present if additional information is required.

name	format	description
RECTYPE	I	is the record type code = 32
NDEX	I	specifies which Level 2 dimension is being described. Counting begins at 0 or 1 as specified in the TEST.SPECB.IDXSTART field. Note that this index does not refer to an index position of an actual data array but is instead a pointer into the DIMSPEC2.INDEX field. See the discussion in Section 2.3.2
DUPNUM	I	is the number of DESCRIP records for this dimension
DESSUP	I	is the number of floating-point values in the SVALS field of the following DESCSUP record. If zero, then there is no DESCSUP record for this dimension
DESFMT	I	is the descriptor format code. This code is explained in Section A.1.1
DESTYPE	I	is the quantity code for the physical quantity that the dimension represents. This code is explained in Section A.1.3
UNITS	I	is the units code for the physical quantity that the dimension represents. This code is explained in Section A.1.4
STORG	I	specifies how the values of the descriptor are stored in the DESCVAL record which follows. The possible values are: 0 = an explicit array of DIMSPEC2.GPTNUM values 1 = an implicit specification consisting of a (low-value, interval) pair 2 = an implicit specification consisting of a (low-value, high-value) pair
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.13 DESCRIP3

For each Level 3 dimension (i =1,...,OBJDESC.NDIM3) there must be DIMSPEC3.DESNUM[i] records. The descriptor records may be in any order, but it is recommended that they be in the order 1 through OBJDESC.NDIM3. This record is considered the authoritative source of information for this dimension. A parallel definition can be supplied in the DESCRIP records. There must be a DESCVAL record for each DESCRIP3 record, to specify the grid points along that dimension. An optional DESCSUP record may be present if additional information is required. The START and END fields indicate which Level 0, Level 1, and Level 2 dimensions this particular Level 3 dimension applies to. For example, the user may have observations that have been integrated in height. The data occur over spatial locations (Level 1) and time (Level 2). The START and END fields will range from (0, 0, 0) to (-1, -1, -1) to cover the entire range of all Level 0–Level 2 dimensions. The number of grid points, GPTNUM, will be the number of height levels that were integrated over. Of course, different DESCRIP3 records referring to different Level 3 dimensions are allowed to have their ranges overlap, since different averaging methods can be applied to the same subset of data.

name	format	description
RECTYPE	I	is the record type code = 33
DEXSORT	I	**NDEX** (bits 0-15) is an index that determines which Level 3 dimension is being described. Counting begins at 0 or 1 as specified in the TEST.SPECB.IDXSTART field. See the discussion in Section 2.3.2 **RECSORT** (bits 16-31) is the index for sorting DESCRIP3 records with the same NDEX field. See the discussion in Section 2.3.2
START	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record begins to apply. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record stops applying. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. A value of -1 indicates the last grid point of that dimension. Counting begins with 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
GPTNUM	I	is the number of grid points along this dimension

AVGCOD	I	is the averaging method code 0 = instantaneous 1 = arithmetic mean (discrete sum) 2 = continuous integration 3 = continuous integration over weighting
DUPNUM	I	is the number of DESCRIP records for this dimension
DESSUP	I	is the number of floating-point values in the SVALS field of the following DESCSUP record. If zero, then there is no DESCSUP record for this dimension
DESFMT	I	is the descriptor format code. This code is explained in Section A.1.1
DESTYPE	I	is the quantity code for the physical quantity that the dimension represents. This code is explained in Section A.1.3
UNITS	I	is the units code for the physical quantity that the dimension represents. This code is explained in Section A.1.4
STORG	I	specifies how the values of the descriptor are stored in the DESCVAL record which follows. The possible values are: 0 = an explicit array of GPTNUM values 1 = an implicit specification consisting of a (low-value, interval) pair 2 = an implicit specification consisting of a (low-value, high-value) pair
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.14 DESCRIP

For each Level j ($j = 1, 2, 3$) dimension there will be DIMSPECj.DUPNUM DESCRIP records. The descriptor records may be in any order, but it is recommended that they be in order of Level 1: 1 through DIMSPEC1.DUPNUM, followed by Level 2: 1 through DIMSPEC2.DUPNUM, followed by Level 3: 1 through DIMSPEC3.DUPNUM. The information in this record type is considered to be an alternative description of the dimension it parallels. There must be a DESCVAL record for each DESCRIP record, to specify the grid points along that dimension. An optional DESCSUP record may be present if additional information is required.

name	format	description
RECTYPE	I	is the record type code = 34
LEVEL	I	is the dimensional level
DEXSORT	I	**NDEX** (bits 0-15) is an index that corresponds to the same field in the associated DIMSPEC1, DIMSPEC2, or DIMSPEC3 record. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2 **RECSORT** (bits 16-31) is an index that corresponds to the same field in the associated DIMSPEC1, DIMSPEC2, or DIMSPEC3 record. See the discussion in Section 2.3.2
DESSUP	I	is the number of floating-point values in the SVALS field of the following DESCSUP record. If zero, then there is no DESCSUP record for this dimension
DESFMT	I	is the descriptor format code. This code is explained in Section A.1.1
DESTYPE	I	is the quantity code for the physical quantity that the dimension represents. This code is explained in Section A.1.3
UNITS	I	is the units code for the physical quantity that the dimension represents. This code is explained in Section A.1.1
STORG	I	specifies how the values of the descriptor are stored in the DESCVAL record which follows. The possible values are: 0 = an explicit array of DESCRIP1.GPTNUM, DIMSPEC2.GPTNUM, or DESCRIP3.GPTNUM values 1 = an implicit specification consisting of a (low-value, interval) pair 2 = an implicit specification consisting of a (low-value, high-value) pair
DINDEX	I	index serving as an identifier (within a given DEXSORT and LEVEL) of the Descgroup record group begun by this record. Values must be greater than zero. (A value of zero would imply that this is a DESCRIP1, DESCRIP2, or DESCRIP3 record instead of DESCRIP.)

RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.15 DESCVAL

For each DESCRIP or DESCRIPj ($j = 1, 2, 3$) record there will be one DESCVAL record. This record type specifies the grid points along the dimension specified.

name	format	description
RECTYPE	I	is the record type code = 35
DLEVEL	I	specifies the dimension level and, if this DESCVAL record is associated with a DESCRIP record (as opposed to DESCRIP1, DESCRIP2, or DESCRIP3), then the DINDEX of that DESCRIP record as well. The LEVEL bit field must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record **LEVEL** (bits 0-1) the dimensional level **DINDEX** (bits 2-31) if zero, this record corresponds to a DESCRIP1, DESCRIP2, or DESCRIP3 record; otherwise, it goes with the DESCRIP record whose LEVEL, DEXSORT and DINDEX fields match this record's
DEXSORT	I	**NDEX** (bits 0-15) is the dimension index. Must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record **RECSORT** (bits 16-31) is the record index. Must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record
AVALS	?*	is an array of values of the descriptor for each grid point of the dimension specified. These values are formatted as in the DESCRIP.DESFMT and DESCRIP.STORG or DESCRIP[LEVEL].DESFMT and DESCRIP[LEVEL].STORG fields

2.4.16 DESCSUP

For each DESCRIP or DESCRIPj ($j = 1, 2, 3$) records there will be one DESCSUP record if specified in DESCRIP.DESSUP or DESCRIPj.DESSUP. This record type specifies the supplementary information about the specified dimension.

name	format	description
RECTYPE	I	is the record type code = 36
DLEVEL	I	specifies the dimension level and, if this record is associated with a DESCRIP record (as opposed to DESCRIP1, DESCRIP2, or DESCRIP3), then the DINDEX of that DESCRIP record as well. The LEVEL bit field must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record **LEVEL** (bits 0-1) the dimensional level **DINDEX** (bits 2-31) if zero, this record corresponds to a DESCRIP1, DESCRIP2, or DESCRIP3 record; otherwise, it goes with the DESCRIP record whose LEVEL, DEXSORT and DINDEX fields match this record's
DEXSORT	I	**NDEX** (bits 0-15) is the dimension index. Must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record **RECSORT** (bits 16-31) is the record index. Must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record
CODE	I	is the supplemental code which specifies the meaning of the SVALS. This code is discussed in Section A.1.6
SVALS	F*	is an array of floating-point values containing supplementary dimensional information. The length of the array must be DESCRIP.DESSUP or DESCRIP[LEVEL].DESSUP

2.4.17 BADVAL

This record type specifies how bad or missing data is flagged. There will be OBJDESC.NBADS BADVAL records, and no BADVAL records will be present if OBJDESC.NBADS is zero. Each component of the data will have an associated bad-data flag (VALUE). Multiple BADVAL records can exist covering overlapping ranges in the Level 1 and Level 2 dimensions; each record is indexed and will be processed in order of their RECSORT fields. Note that it is possible to specify different bad-data flags over different ranges of the Level 1 and Level 2 dimensions. For example, a data set of wind vectors might have the bad-data flags depend on the pressure level, but be independent of latitude and longitude. Note also that each flag specification would consist of two values: one each for the East–West component and North–South component of the wind vector.

name	format	description
RECTYPE	I	is the record type code = 40
RECSORT	I	is the index for sorting the records. Lower valued records will be processed before higher valued records. See the discussion in Section 2.3.2
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
START	I*	is an array of Level 1 and Level 2 dimension grid points over which this record begins to apply. The first OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an array of Level 1 and Level 2 dimension grid points over which this record stops applying. The first OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. A value of -1 indicates the last grid point of that dimension. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
VALUE	?*	is an array with the length equal to the multiplicative sum of the number of grid points of all the Level 0 dimensions (DIMSPEC0.GPTNUM). They are bad-data flags for each data component and are formatted according to DESCRIP0.DATFMT, i.e., they are in the same numeric (or string) format as the component data values

2.4.18 PROCSPEC

This record type specifies processing codes for the data. The data object will contain OBJDESC.NPROCS records of this record type, and no records of this type if OBJDESC.NPROCS is zero. The records will be processed in order of their RECSORT fields. Multiple PROCSPEC records can specify an overlap of grid points in their START and END fields, since there can be different processing codes for the same subset of data.

name	format	description
RECTYPE	I	is the record type code = 50
RECSORT	I	is the index for sorting the records. Lower valued records will be processed before higher valued records. See the discussion in Section 2.3.2
START	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record begins to apply. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record stops applying. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. A value of -1 indicates the last grid point of that dimension. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
CODE	I	is the processing code. This code is explained in Section A.2.3
PRCNUM	I	is the number of values in the PROCVAL.INFO field. These values have meaning in terms of the CODE field, and PRCNUM should match the number of values implied by the CODE field. If no values are expected, then this should be set to zero, and there will be no PROCFORM or PROCVAL records associated with this PROCSPEC record
NDUPS	I	is the number of PROCDUP records associated with this PROCSPEC record
RESERVE0	I	is reserved for future expansion

2.4.19 PROCFORM

This record type specifies the format of the values of the processing information in PROCVAL.INFO. There must be one record for each PROCSPEC record unless PROCSPEC.PRCNUM is zero. There are PROCSPEC.PRCNUM elements in the PRCFMT array: one for each value in the PROCVAL.INFO field.

name	format	description
RECTYPE	I	is the record type code = 51
RECSORT	I	is the same index as the corresponding PROCSPEC.RECSORT field
PRCFMT	I*	is the format code of the processing values in the PROCVAL.INFO field. These codes are explained in Section A.1.1

2.4.20 PROCVAL

This record type specifies the values associated with the processing information. There must be one record for each PROCSPEC record, unless PROCSPEC.PRCNUM is zero. There are PROCSPEC.PRCNUM elements in the INFO array.

name	format	description
RECTYPE	I	is the record type code = 52
RECSORT	I	is the same index as the corresponding PROCSPEC.RECSORT field
INFO	I*	contains the values needed for interpreting the processing codes. Their formats are specified in PROCFORM.PRCFMT

2.4.21 PROCDUP

This record type specifies additional Level 0, Level 1, and Level 2 dimensional (grid points) ranges over which the same processing code and associated values apply as were specified in a previous PROCSPEC record. There will be PROCSPEC.NDUPS records in this record type, and this record type will not be present if PROCSPEC.NDUPS is zero. There can be an overlap of grid points specified in the START and END fields for different records, since there can be different processing codes for the same subset of data.

name	format	description
RECTYPE	I	is the record type code = 53
RECSORT	I	is the same index as the corresponding PROCSPEC.RECSORT field
PINDEX	I	is an index for sorting PROCDUP records associated with a given PROCSPEC record. Lower valued records will be processed before higher valued records. See the discussion in Section 2.3.2
START	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record begins to apply. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record stops applying. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. A value of -1 indicates the last grid point of that dimension. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2

2.4.22 AUXSPEC

This record specifies what kind of auxiliary information follows in an AUXVAL record. The data object will contain OBJDESC.NAUX records of this record type, and no records of this type if OBJDESC.NAUX is zero. Multiple Auxgroup record groups can exist covering overlapping ranges in the START and END fields. Each record will be processed in order of the RECTYPE index, higher-numbered records taking precedence over the lower-numbered records where conflicts exist.

name	format	description
RECTYPE	I	is the record type code = 80
RECSORT	I	is the index for sorting the records. Lower valued records will be processed before higher valued records. See the discussion in Section 2.3.2
NUMREF	I	the number of dimensions the auxiliary information refers to.
NUMAPP	I	is the number of dimensions the auxiliary information applies to.
START	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points beginning with which this Auxgroup start to be valid. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points beyond which this Auxgroup is no longer valid. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. A value of -1 indicates the last grid point of that dimension. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
AUXFMT	I	is the data format code. This code is explained in Section A.1.1
AUXTYPE	I	is the quantity code for the auxiliary information. Note that the full meaning of this code may depend upon the quantity code of the data or dimensions to which the auxiliary information refers. This code is explained in Section A.1.3
UNITS	I	is the units code for the auxiliary information. This code is explained in Section A.1.4. If this field is zero, then there will either be no units or the units will be the same as the data and dimensions that the auxiliary information refers to
NUMSUP	I	is the number of values in the AUXSUP.SVALS field which follows. If zero, then there will be no AUXSUP record
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.23 AUXRANGE

This record type specifies the dimensions to which an Auxgroup refers and over which it applies.

name	format	description
RECTYPE	I	is the record type code = 81
RECSORT	I	is the same index as the corresponding AUXSPEC.RECSORT field
REFLEV	I*	is an AUXSPEC.NUMREF-element array of the dimension level (and, if this Auxgroup is associated with a DESCRIP record dimensional description, the DINDEX of that DESCRIP record as well) of each dimension to which this Auxgroup refers. The LEVEL bit field must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record. Each element of this array has the structure: **LEVEL** (bits 0-1) the level of the dimension **DINDEX** (bits 2-31) if zero, the dimension referred to is defined by a DESCRIP1, DESCRIP2, or DESCRIP3 record; otherwise, it is defined by the DESCRIP record whose LEVEL, NDEX and DINDEX fields match this record's REFLEV.LEVEL, REFNDEX, and REFLEV.DINDEX fields
REFNDEX	I*	is an AUXSPEC.NUMREF-element array of the indices of the dimensions (within their respective Levels) to which this Auxgroup refers. Note that these refer to the NDEX fields of the DESCRIP1, DESCRIP2, DESCRIP3, or DESCRIP records, and not to indices of the actual data array
APPLEV	I*	is an AUXSPEC.NUMAPP-element array of the dimension level (and, if this Auxgroup is associated with a DESCRIP record dimensional description, the DINDEX of that DESCRIP record as well) of each dimension over which this Auxgroup applies. The LEVEL bit field must be the same as in the corresponding DESCRIP or DESCRIP[LEVEL] record. Each element of this array has the structure: **LEVEL** (bits 0-1) the level of the dimension **DINDEX** (bits 2-31) if zero, the dimension over which the Auxgroup applies is defined by a DESCRIP1, DESCRIP2, or DESCRIP3 record; otherwise, it is defined by the DESCRIP record whose LEVEL, NDEX and DINDEX fields match this record's APPLEV.LEVEL, APPNDEX, and APPLEV.DINDEX fields
APPNDEX	I*	is an AUXSPEC.NUMAPP-element array of the indices of the dimensions (within their respective Levels) over which this Auxgroup applies. Note that these refer to the NDEX

		fields of the DESCRIP1, DESCRIP2, DESCRIP3, or DESCRIP records, and not to indices of the actual data array. Also, note that a reference dimension (as given by the AUXRANGE.REFLEV and AUXRANGE.REFNDEX fields) may be repeated in the list of application dimensions. This means that the auxiliary values vary over a quantity to which they refer; for example, if the auxiliary values are uncertainties in a vector wind field, then the North–South component might vary from the East–West component. The application dimensions, together with the START–END pairs from the AUXSPEC record, determine which component this Auxgroup refers to. If a reference dimension is *not* repeated among the application dimensions, then the Auxgroup is considered to refer to the entire range of that reference dimension
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.24 AUXVAL

This record type specifies the auxiliary information values associated with an Auxgroup. There must be one record for each AUXSPEC record.

name	format	description
RECTYPE	I	is the record type code = 82
RECSORT	I	is the same index as the corresponding AUXSPEC.RECSORT field
AVALS	?*	an array of auxiliary data values. Their format is given by AUXSPEC.AUXFMT. These values constitute an array whose size and shape corresponds to the AUXRANGE.APPLEV.LEVEL, and AUXRANGE.APPNDEX fields. That is, the first index into this array corresponds to the AUXRANGE.APPNDEX[1]'th dimension at the AUXRANGE.APPLEV[1]'th level, and so on through all the application dimensions. The size of the AVALS field along each of its array dimensions is given by the range specified by those elements of the AUXSPEC.START and AUXSPEC.END fields which is associated with the application dimension to which the AVALS array dimension corresponds. In the case of a START–END pair which covers the entire range of a dimension, the size will be the number of grid points along that application dimension.

2.4.25 AUXSUP

This record type specifies any supplementary values associated with this Auxgroup. There must be zero or one of these records for each AUXSPEC record. The meaning of the supplemental values depends upon the AUXSUP.CODE field.

name	format	description
RECTYPE	I	is the record type code = 82
RECSORT	I	is the same index as the corresponding AUXSPEC.RECSORT field
RESERVE0	I	is reserved for future expansion
CODE	I	is the supplemental code which specifies the meaning of the SVALS. This code is discussed in Section A.1.6
SVALS	F*	an array of AUXSPEC.NUMSUP values used to help interpret the auxiliary information.

2.4.26 PAKSPEC

This record type specifies how the data are packed. The data object will contain OBJDESC.NPACKS records of this record type, and no records of this type if OBJDESC.NPACKS is zero. The records will be processed in order of their RECSORT fields. There can be an overlap of grid points specified in the START and END fields, but this use is strongly discouraged.

name	format	description
RECTYPE	I	is the record type code = 60
RECSORT	I	is the index for sorting the records. Lower valued records will be processed before higher valued records. See the discussion in Section 2.3.2
START	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record begins to apply. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
END	I*	is an array of Level 0, Level 1, and Level 2 dimension grid points over which this record stops applying. The first OBJDESC.NDIM0 elements correspond to the Level 0 dimensions. The middle OBJDESC.NDIM1 elements correspond to the Level 1 dimensions. The last OBJDESC.NDIM2 elements correspond to the Level 2 dimensions. A value of -1 indicates the last grid point of that dimension. Counting begins at 0 or 1 as specified in TEST.SPECB.IDXSTART. See the discussion in Section 2.3.2
CODE	I	is the packing code. This code is explained in Section A.1.5
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion
DPAKFMT	I	is the format of the packed data values in the PAKDAT.VALS field. (After unpacking, the data will have the format as specified in DESCRIP0.DATFMT.) See Section A.1.1
PAKNUM	I	is the number of values in the PAKVAL.INFO field. PAKNUM should match the number of values implied by the CODE field. If there are no values, then this should be set to zero, and there will be no PAKFORM record and no PAKVAL.INFO field

2.4.27 PAKFORM

This record type specifies the format of the values of the packing information in PAKVAL.INFO. There must be one record for each PAKSPEC record, unless PAKSPEC.PAKNUM is zero. There are PAKSPEC.PAKNUM elements in the PAKFMT array, one for each value in the PAKVAL.INFO field.

name	format	description
RECTYPE	I	is the record type code = 61
RECSORT	I	is the same index as the corresponding PAKSPEC.RECSORT field
PAKFMT	I*	is the format code of the packing information values in the PAKVAL.INFO field. These codes are explained in Section A.1.1

2.4.28 PAKVAL

This record type specifies the values for the processing information. There must be one record for each PAKSPEC record, even if PAKSPEC.PAKNUM is zero, since the PAKBAD value must still be specified. There are PAKSPEC.PAKNUM elements in the INFO array, and this number may be zero.

name	format	description
RECTYPE	I	is the record type code = 62
RECSORT	I	is the same index as the corresponding PAKSPEC.RECSORT field
PAKBAD	?	is the bad-data flag value of the packed data in the PAKDAT.VALS field. The unpacked data will use BADVAL.VALUE as the bad-data flag. The format of this field is specified in PAKSPEC.DPAKFMT
INFO	I*	contains the values needed for interpreting the packing codes. These values have meaning as implied by the PAKSPEC.CODE field. The format is specified in PAKFORM.PAKFMT. If there are no values needed for the packing method, then this field will not exist, and PAKSPEC.PAKNUM will be set to zero

2.4.29 COMPSPEC

This record type specifies how the data are compressed. The data object will contain OBJDESC.CMPNUM records of this record type, and no records of this type if OBJDESC.CMPNUM is zero.

name	format	description
RECTYPE	I	is the record type code = 70
CODE	I	is the compression code. This code is explained in Section A.1.7
COMPNUM	I	is the number of values in the COMPVAL.INFO field. COMPNUM should match the number of values implied by the CODE field. If there are no values, then this should be set to zero, and there will be no COMPFORM or COMPVAL records
RESERVE0	I	is reserved for future expansion
RESERVE1	I	is reserved for future expansion

2.4.30 COMPLEN

This record type specifies the length in bytes of each COMPDAT record (of which there are OBJDESC.CMPNUM). Thus, languages which need to know how long a record is before it is read will be able to tell how long the COMPDAT records are.

name	format	description
RECTYPE	I	is the record type code = 71
LENGTHS	I*	is an array of the lengths in bytes of the COMPDAT.VALS field in each COMPDAT record. (Note, then that these lengths do *not* include the length of the RECTYPE field in those records

2.4.31 COMPFORM

This record type specifies the format of the values of the compression information (i.e., parameters) in COMPVAL.INFO. There are COMPSPEC.COMPNUM elements in the COMPFMT field array, one for each value in the COMPVAL.INFO field. If COMPSPEC.COMPNUM is zero, this record will not exist.

name	format	description
RECTYPE	I	is the record type code = 72
COMPFMT	I*	is the format code of the compression information values in the COMPVAL.INFO field. These codes are explained in Section A.1.1

2.4.32 COMPVAL

This record type specifies the format of the values of the compression information. There must be one record for each COMPSPEC record, unless COMPSPEC.COMPNUM is zero. There are COMPSPEC.COMPNUM elements in the INFO array.

name	format	description
RECTYPE	I	is the record type code = 73
INFO	I*	contains the values needed for interpreting the compression codes. These values have meaning as implied by the COMPSPEC.CODE field. The format is specified in COMPFORM.COMPFMT

2.4.33 REGDAT

This record type contains the actual data values if the data have not been packed or compressed. The number of REGDAT records is the multiplicative summation of the DIMSPEC2.GPTNUM values. The number of elements in in the VALS field of each record is the multiplicative summation of all the DESCRIP1.GPTNUM and DIMSPEC0.GPTNUM values. The format of the data in the VALS field is given in DESCRIP0.DATFMT.

name	format	description
RECTYPE	I	is the record type code = 100
VALS	?*	contains the data values

2.4.34 PAKDAT

This record type contains the packed data values if the data have been packed but not compressed. The number of PAKDAT records is the multiplicative summation of the DIMSPEC2.GPTNUM values. The number of elements in the VALS field of each record is the multiplicative summation of all the DESCRIP1.GPTNUM and DIMSPEC0.GPTNUM values. The format of the VALS field is given in PAKSPEC.DPAKFMT.

name	format	description
RECTYPE	I	is the record type code = 110
VALS	?*	contains the packed data values

2.4.35 COMPDAT

This record type contains the compressed data values. There are OBJDESC..CMPNUM of these COMPDAT records. The VALS field contains a byte array of compressed values; the lengths of the i'th field is given by the i'th element of the COMPLEN.LENGTHS array. When this field is uncompressed, it will contain the values as specified in a series of PAKDAT or REGDAT records without those records' RECTYPE fields.

name	format	description
RECTYPE	I	is the record type code = 120
VALS	?*	contains the compressed data values

Chapter 3

Discussion and Examples of the Format

In this chapter examples are presented as an aid to understanding the structures defined and explained in Chapter 2. Space does not permit listing entire example programs here; therefore, isolated fragments of code are used to illustrate concepts. These fragments are written in Fortran, C code, and IDL (Interactive Data Language, from Research Systems, Inc., of Boulder, Colorado). The reader who is unfamiliar with one or more of these languages should still be able to follow the examples. Note that the programming statements shown are merely illustrative—data declarations, error checking, *et cetera*, are omitted here, although such things would be necessary in any working program. In other words, these code fragments are examples to be studied, not templates to be copied slavishly.

The first section, for example, deals with writing the TEST record in several languages. In Section 3.2, examples illustrate how dimensions are specified in a df dataset. In Section 3.3, the use of the START and END fields found in various records is shown. In Section 3.4, the information in the AUDIT record is discussed. Finally, in Section 3.5, various examples are provided showing how to specify auxiliary information.

3.1 TEST records

We present two code fragments for writing out the information in the TEST record. The first example is coded in C, the second in Fortran. These two examples will illustrate the differences in the TEST record when using different programming languages.

3.1.1 Silicon Graphics Iris workstation, using C

```
        /* declare the test record as a 24-
element byte array */
        char testrec[24];
```

```
        /* MAGIC: we load in the magic number */
        testrec[0] = 0x47 ;
        testrec[1] = 0xf3 ;
        testrec[2] = 0x46 ;
        testrec[3] = 0xe3 ;

        /* MACHID: SGI Unix workstation */
        testrec[4] = 2 ;

        /* NUMOBJECTS: We want to write out one ob-
ject, so
           this can be either 0 or 1 */
        testrec[5] = 0 ;

        /* SPECA: We are using ASCII charac-
ters, and SGIs are
           big-endian machines */
        testrec[6] = 0x01 ;

        /* RESERVED1 */
        testrec[7] = 0 ;

        /* RECHDR: This is a C program using normal I/O
           functions, so the file will be a pure stream of
           bytes */
        testrec[8] = 1 ;

        /* RESERVED2 */
        testrec[9] = 0 ;

        /* SPECB: This is C, so we are using row-
major array
           ordering, and our array in-
dices start with zero */
        testrec[10] = 0x00 ;

        /* SLEN: a short integer is 16 bits on an SGI */
        testrec[11] = 16;

        /* LLEN: a long integer is 32 bits */
        testrec[12] = 32;

        /* FLEN: a floating-point number is 32 bits */
        testrec[13] = 32;
```

```
        /* DLEN: a double precision floating-point num-
ber is
           64 bits */
        testrec[14] = 64;

        /* FPFORM: SGI machines use IEEE floating-point
           formats */
        testrec[15] = 0x11;

        /* RESERVED3 */
        testrec[16] = 0 ;
        /* RESERVED4 */
        testrec[17] = 0 ;
        /* RESERVED5 */
        testrec[18] = 0 ;
        /* RESERVED6 */
        testrec[19] = 0 ;
        /* RESERVED7 */
        testrec[20] = 0 ;
        /* RESERVED8 */
        testrec[21] = 0 ;
        /* RESERVED9 */
        testrec[22] = 0 ;
        /* RESERVED10 */
        testrec[23] = 0 ;

        /* now write out the record */
        (void) fwrite( testrec, sizeof(char), 24, outfile);
```

3.1.2 VAX running VMS using Fortran

```
C234567
C      We declare the test record as an array of bytes
       INTEGER*1 TESTREC(24)
C
C      MAGIC: We load in the magic number (in decimal)
       TESTREC(1) = 71
       TESTREC(2) = -13
       TESTREC(3) = 70
       TESTREC(4) = -29
C
C      MACHID:  VAX/VMS
       TESTREC(5) = 1
C
C      NUMOBJECTS: We will write only one object,
```

```
C      so this can be 0 or 1.
       TESTREC(6) = 0
C
C      SPECA: We use ASCII, and the VAX is a little-
endian
C      machine
       TESTREC(7) = 13
C
C      RESERVED1
       TESTREC(8) = 0
C
C      RECHDR: We are using VAX Fortran Record Headers
C      (I.e., segmented variable length records)
       TESTREC(9) = 2
C
C      RESERVED2
       TESTREC(10) = 0
C
C      SPECB: In Fortran, arrays are in column-major or-
der,
C      and array indices start from 1.
       TESTREC(11) = 3
C
C      SLEN: On a VAX, a short integer is 16 bits
       TESTREC(12) = 16
C
C      LLEN: On a VAX, a long integer is 32 bits
       TESTREC(13) = 32
C
C      FLEN: On a VAX, a floating-point number is 32 bits
       TESTREC(14) = 32
C
C      DLEN: On a VAX, a double precision floating-point
C      number is 64 bits
       TESTREC(15) = 64
C
C      FPFORM: We are using VAX format floating-
point numbers
C      (VAX D for double precision).
       TESTREC(16) = 34
C
C      MORE RESERVED...
       TESTREC(17) = 0
       TESTREC(18) = 0
       TESTREC(19) = 0
       TESTREC(20) = 0
```

```
      TESTREC(21) = 0
      TESTREC(22) = 0
      TESTREC(23) = 0
      TESTREC(24) = 0
C
C     Now write it all out
      WRITE(LUNIT) TESTREC
C
```

3.2 Dimensional Levels

Following are examples of specifying dimensions for several different data objects. We first present an example specifying a scalar data object with uncomplicated dimensions. In the second example, we add the complications of using a vector data object and dimension indices which are defined in a different order than the data array's indices. In the third example, a tensor data object is used, to make things even more complicated.

These first three examples consider only cases where the Level 1 dimensions do not vary over the Level 2 dimensions (i.e., all of the data records are the same size.) In the last two examples, we present cases where they do vary. The first shows a complex data object, where two related quantities are combined as components of a vector. Finally, the last example illustrates the use of nonauthoritative dimension descriptors.

3.2.1 Example 1. Scalar (Temperature)

Consider a data object consisting of temperatures measured over a longitude-latitude grid and a set of standard atmospheric pressure levels. There may be one or more observations in time; we will suppose that we have only one set of data, valid for 22 July 1991. The longitudes will run every 5 degrees starting from 0, and the latitudes will run every 2 degrees starting from -90. We further suppose that the data will lie on 6 pressure surfaces: 1000, 850, 700, 500, 250, and 100 mb.

Temperature is a scalar, and so the data object will have a single Level 0 dimension having one element. A natural way to think of the data is as a set of six two-dimensional arrays. That is, we will have two Level 1 dimensions, longitude and latitude, and at least one Level 2 dimension: pressure. Because we have data for only one time, there are several ways we could indicate this in the file: putting a time stamp in an INFOSPEC or COMMENT record, adding it in as a Level 1 or Level 2 dimension, or making time a Level 3 dimension whose average type is the oxymoronic "instantaneous." We choose to have time as a second Level 2 dimension with one grid point.

We choose to write the data as a $72 \times 91 \times 6$ (i.e, longitude, latitude, pressure) array.

If our indices start numbering at 0, then we have the following record fields:

OBJDESC.	NDIM0 = 1
	NDIM1 = 2
	NDIM2 = 2
	NDIM3 = 0

DIMSPEC0.	INDEX = -1	
	GPTNUM = 1	
DIMSPEC1.	INDEX = (0, 1)	
	DESNUM = (1, 1)	
DIMSPEC2.	INDEX = (2, 3)	
	GPTNUM = (6, 1)	
DESCRIP0.	DATFMT = single precision floating-point	
	VARTYPE = temperature	
	UNITS = K	
DESCRIP1.	DEXSORT = 0	
		NDEX = 0
		RECSORT = 0
	START = (0, 0)	
	END = (-1, -1)	
	GPTNUM = 72	
	DUPNUM = 0	
	DESSUP = 0	
	DESFMT = long integer	
	DESTYPE = longitude	
	UNITS = degree	
	STORG = 1	
DESCVAL.	DLEVEL = 1	
		LEVEL = 1
		DINDEX = 0
	DEXSORT = 0	
		NDEX = 0
		RECSORT = 0
	AVALS = (0, 5)	
DESCRIP1.	DEXSORT = 1	
		NDEX = 1
		RECSORT = 0
	START = (0, 0)	
	END = (-1, -1)	
	GPTNUM = 91	
	DUPNUM = 0	
	DESSUP = 0	
	DESFMT = long integer	
	DESTYPE = latitude	
	UNITS = degree	
	STORG = 2	
DESCVAL.	DLEVEL = 1	
		LEVEL = 1
		DINDEX = 0
	DEXSORT = 1	
		NDEX = 1
		RECSORT = 0

	AVALS = (-90, 90)
DESCRIP2.	NDEX = 0
	DUPNUM = 0
	DESSUP = 0
	DESFMT = single precision floating-point
	DESTYPE = pressure
	UNITS = mb
	STORG = 0
DESCVAL.	DLEVEL = 2
	LEVEL = 2
	DINDEX = 0
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	AVALS = (1000., 850., 700., 500., 250., 100.)
DESCRIP2.	NDEX = 1
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = time
	UNITS = day of year
	STORG = 0
DESCVAL.	DLEVEL = 2
	LEVEL = 2
	DINDEX = 0
	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	AVALS = 203

In IDL, then, code to define the dimensions of this data object would look like this:

```
    ; set up the relevant elements in the OBJDESC record
    RECTYPE = long(1)
    NDIM0 = long(1)
    NDIM1 = long(2)
    NDIM2 = long(2)
    NDIM3 = long(0)
    .
    .
    .
    writeu,log_file_unit,   RECTYPE, vartype, re-
served    $
    , NDIM0, NDIM1, NDIM2, NDIM3, re-
served, isource       $
```

```
    , reserved, reserved, naudit, ninfo, ncomms, com-
cod   $
    , reserved, nbads, reserved, nprocs, re-
served, npacks $
    , reserved, reserved, cmpnum, reserved, re-
served      $
    , reserved

    .
    .
    .

    ; set up the DIMSPEC0 record
    RECTYPE = long(20)
    ;  Since Temperature is a scalar, there is no ar-
ray index
    ,  for this to point to.
    INDEX = long(-1)
    ;  Since temperature is a scalar, there is only one grid
    ;  point.
    GPTNUM = long(1)

    writeu,log_file_unit,  RECTYPE, reserved, reserved $
    , INDEX, GPTNUM

    ; set up the DIMSPEC1 record
    RECTYPE = long(21)
    ; longitude and latitude will be the first and second
    ; dimensions of the data array read in.
    INDEX = long( [ 0, 1 ] )
    ; we will specify only one DESCRIP1 record for each
    ; dimension
    DESNUM = long( [ 1, 1 ] )

    writeu,log_file_unit,  RECTYPE, reserved, reserved $
    , INDEX, DESNUM

    ; set up the DIMSPEC2 record
    RECTYPE = long(22)
    ; pressure and time will be the third and fourth
    ; dimensions of the data array read in.
    INDEX = long( [ 2, 3 ] )
    ; There will be five pressure levels and one time
    GPTNUM = long( [ 6, 1 ] )
```

```
writeu,log_file_unit,  RECTYPE, reserved, reserved $
, INDEX, GPTNUM

; (There is no DIMSPEC3 record.)

; set up the DESCRIP0 record
RECTYPE = long(30)
; data are floating-point single precision numbers
DATFMT = long(67108864)
; data are temperatures
; Note: in actual working code, one would call here a
; function which would convert the string "Temperature"
; to its quantity ID code 16777216:
; vartype = name_to_number("temperature")
VARTYPE = long(16777216)
; the temperatures are in Kelvin
UNITS = long(681050112)

writeu,log_file_unit, RECTYPE, reserved, reserved $
, DATFMT, VARTYPE, UNITS

; set up the longitude DESCRIP1 record
RECTYPE = long(31)
; This describes the first Level 1 dimension
NDEX = long(0)
; This covers the entire range of both Level 2 dimensions
START = long( [ 0, 0 ] )
; (Note: END is a reserved word in IDL, so we use ENDIT
; here)
ENDIT = long( [ -1, -1 ] )
; There are 72 longitudes
GPTNUM = long(72)
; No duplicate records
DUPNUM = long(0)
; No supplemental information
DESSUP = long(0)
; longitudes are long integers
DESFMT = long(51445760)
; the quantity id for longitude is 17838080
DESTYPE = long(17838080)
; The units code for degrees is 1745355010
UNITS = long(1745355010)
; We will store this as a an implicit
; (low-value, interval) pair
STORG = long(1)
```

```
writeu,log_file_unit, RECTYPE, NDEX, START, ENDIT $
, GPTNUM, DUPNUM, DESSUP, DESFMT, DESTYPE, UNITS $
, STORG, reserved, reserved

; set up the longitude DESCVAL record
RECTYPE = long(35)
; This belongs to Level 1, dimension number 1
LEVEL = long(1)
NDEX = long(0)
; We start at 0 longitude and increase by 5 degrees
; thereafter
AVALS = long([ 0, 5 ] )

writeu,log_file_unit, RECTYPE, LEVEL, NDEX, AVALS

; Now do the latitude DESCRIP1 and DESCVAL records
; (This is more terse, to show that you do not really
; need a lot of code to write these records;  as an
; exercise, the reader may wish to interpret the
; following).  The quantity code for latitude is
; 17838096
writeu,log_file_unit, long(31), long(1), long([0,0]) $
, long([-1,-1]), long(91), long(0), long(0)          $
, long(51445760), long(17838096), long(1745355010)   $
, long(2), reserved, reserved
writeu,log_file_unit, long(35), long(1), long(1) $
, long([ -90, 90 ])

; Do the pressure DESCRIP2 record
RECTYPE = long(32)
; This describes the first Level 2 dimension
NDEX = long(0)
; No duplicate records
DUPNUM = long(0)
; No supplemental information
DESSUP = long(0)
; pressures are floating-point numbers
DESFMT = long(67108864)
; the quantity id for pressure is 16781312
DESTYPE = long(16781312)
; The units code for millibars is 1081593921
UNITS = long(1081593921)
; We will store this as a an explicit array
STORG = long(0)
```

```
writeu,log_file_unit, RECTYPE, NDEX, DUPNUM, DESSUP $
, DESFMT, DESTYPE, UNITS, STORG, reserved, reserved

; set up the pressure DESCVAL record
RECTYPE = long(35)
; This belongs to Level 2, dimension number 1
LEVEL = long(2)
NDEX = long(0)
; We list the pressure levels
AVALS = float([1000., 850., 700., 500., 250., 100.] )

writeu,log_file_unit, RECTYPE, LEVEL, NDEX, AVALS

; do the time dimension spec (tersely: an-
other exercise
; for the reader.  The format code for long unsigned
; integers is 50397184, the quantity code for time is
; 131072, and the units code for day number (from
; January 1) is 1615331845.  July 22 was the 203rd
; day of 1991.
writeu,log_file_unit, long(32), long(1), long(0) $
, long(0), long(50397184), long(131072) $
, long(1615331845), long(0), reserved, reserved
writeu,log_file_unit, long(35), long(2), long(1) $
, long(203)
```

3.2.2 Example 2. Vector (Wind)

Consider a data object consisting of a wind vector measured over a longitude-latitude grid and standard atmospheric pressure levels, as in the previous example. The winds are averages over ten years for each day in January. Each wind measurement is a vector having three components (u, v, w), one along each of the dimensions of longitude, latitude, and pressure.

Thus the data have one Level 0 dimension with three elements. Longitude and latitude are Level 1 dimensions as in the previous example, but this time we make pressure a Level 1 dimension, too, instead of Level 2.

We choose to write the data out as a $6 \times 3 \times 91 \times 72 \times 31$ array (i.e., pressure, wind component, latitude, longitude, day of month).

If our indices start numbering at 0, then we have the following record fields:

OBJDESC.	NDIM0 = 1
	NDIM1 = 3
	NDIM2 = 1
	NDIM3 = 1

DIMSPEC0. INDEX = 1
GPTNUM = 3
DIMSPEC1. INDEX = (3, 2, 0)
DESNUM = (1, 1, 1)
DIMSPEC2. INDEX = 4
GPTNUM = 31
DIMSPEC3. DESNUM = 1
DESCRIP0. DATFMT = (single precision floating-point, single precision floating-point, single precision floating-point)
VARTYPE = (u, v, w)
UNITS = (m/s, m/s, cm/s)
DESCRIP1. DEXSORT = 1
NDEX = 1
RECSORT = 0
START = 0
END = -1
GPTNUM = 91
DUPNUM = 0
DESSUP = 0
DESFMT = long integer
DESTYPE = latitude
UNITS = degree
STORG = 2
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 1
NDEX = 1
RECSORT = 0
AVALS = (-90, 90)
DESCRIP1. DEXSORT = 0
NDEX = 0
RECSORT = 0
START = 0
END = -1
GPTNUM = 72
DUPNUM = 0
DESSUP = 0
DESFMT = long integer
DESTYPE = longitude
UNITS = degree
STORG = 1
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 0

NDEX = 0
RECSORT = 0
AVALS = (0, 5)
DESCRIP1. DEXSORT = 2
NDEX = 2
RECSORT = 0
START = 0
END = -1
GPTNUM = 6
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 2
NDEX = 2
RECSORT = 0
AVALS = (1000., 850., 700., 500., 250., 100.)
DESCRIP2. NDEX = 0
DUPNUM = 0
DESSUP = 0
DESFMT = long integer
DESTYPE = time
UNITS = day of month
STORG = 1
DESCVAL. DLEVEL = 2
LEVEL = 2
DINDEX = 0
DEXSORT = 0
NDEX = 0
RECSORT = 0
AVALS = (1, 1)
DESCRIP3. DEXSORT = 0
NDEX = 0
RECSORT = 0
START = (0, 0, 0, 0, 0)
END = (-1, -1, -1, -1, -1)
GPTNUM = 10
AVGCOD = 1
DUPNUM = 0
DESSUP = 0
DESFMT = long integer

```
            DESTYPE = time
            UNITS = year
            STORG = 2
DESCVAL.    DLEVEL = 3
                LEVEL = 3
                DINDEX = 0
            DEXSORT = 0
                NDEX = 0
                RECSORT = 0
            AVALS = (1983, 1992)
```

In IDL, then, the code for specifying the dimensions would look something like this:

```
    ; set up the relevant elements in the OBJDESC record
    RECTYPE = long(1)
    NDIM0 = long(1)
    NDIM1 = long(3)
    NDIM2 = long(1)
    NDIM3 = long(1)
    .
    .
    .
    writeu,log_file_unit,   RECTYPE, vartype, re-
served    $
    , NDIM0, NDIM1, NDIM2, NDIM3, re-
served, isource       $
    , reserved, reserved, naudit, ninfo, ncomms, com-
cod   $
    , reserved, nbads, reserved, nprocs, re-
served, npacks $
    , reserved, reserved, cmpnum, reserved, re-
served      $
    , reserved

    .
    .
    .

    ; set up the DIMSPEC0 record
    RECTYPE = long(20)
    ;  The wind component index is second in the ac-
tual data
    ;  array
    INDEX = long(1)
    ;  There are three components to the wind
    GPTNUM = long(3)
```

```
    writeu,log_file_unit,  RECTYPE, reserved, reserved $
    , INDEX, GPTNUM

    ; set up the DIMSPEC1 record
    RECTYPE = long(21)
    ; longitude, latitude, and pressure are the fourth,
    ; third, and first dimensions of the data ar-
ray read in.
    INDEX = long( [ 3, 2, 0 ] )
    ; we will specify only one DESCRIP1 record for each
    ; dimension
    DESNUM = long( [ 1, 1, 1 ] )

    writeu,log_file_unit,  RECTYPE, reserved, reserved $
    , INDEX, DESNUM

    ; set up the DIMSPEC2 record
    RECTYPE = long(22)
    ; Time will be the fifth dimension of the data ar-
ray read
    ; in.
    INDEX = long( 4 )
    ; There will be 31 times, one for each day of January
    GPTNUM = long( 31 )

    writeu,log_file_unit,  RECTYPE, reserved, reserved $
    , INDEX, GPTNUM

    ; set up the DIMSPEC3 record
    RECTYPE = long(23)
    ; there will be only one DESCRIP3 record for the single
    ; Level 3 dimension
    DESNUM = 1

    writeu,log_file_unit, RECTYPE, reserved, re-
served, DESNUM

    ; set up the DESCRIP0 record
    RECTYPE = long(30)
    ; data are floating-point single precision numbers
    DATFMT = long(67108864, 67108864, 67108864)
    ; data are wind componenets (u, v, w)
    ; Note: in actual working code, one would call here a
    ; function which would convert the strings
    ; "measured u wind", "measured v wind", and
```

```
    ; "measured w wind" to the quantity codes:
    ; 18874368, 18878464, 18882560
    VARTYPE = long(18874368, 18878464, 18882560)
    ; the wind components are in m/s, m/s, cm/s
    UNITS = long(1616347136, 1616347136, 1616347137)

    writeu,log_file_unit, RECTYPE, reserved, reserved $
    , DATFMT, VARTYPE, UNITS

    ; Now do the latitude DESCRIP1 and DESCVAL records
    writeu,log_file_unit, long(31), long(1), long(0)       $
    , long(-1), long(91), long(0), long(0), long(51445760) $
    , long(17838096), long(1745355010), long(2), re-
served  $
    , reserved
    writeu,log_file_unit, long(35),long(1),long(1) $
    , long([ -90, 90 ])

    ; Now do the longitude DESCRIP1 and DESCVAL records
    writeu,log_file_unit, long(31), long(0), long(0)       $
    , long(-1), long(72), long(0), long(0), long(51445760) $
    , long(17838080), long(1745355010), long(1), re-
served  $
    , reserved
    writeu,log_file_unit, long(35), long(1), long(0), $
    , long([ 0, 5 ])

    ; Do the pressure DESCRIP1 record
    RECTYPE = long(31)
    ; This describes the third Level 1 dimension
    NDEX = long(2)
    ; This covers the entire range of the single Level 2
    ; dimension
    START = long(0)
    ENDIT = long(-1)
    ; Six pressure levels
    GPTNUM = 6
    ; No duplicate records
    DUPNUM = long(0)
    ; No supplemental information
    DESSUP = long(0)
    ; pressures are floating-point numbers
    DESFMT = long(67108864)
    ; the quantity id for pressure is 16781312
    DESTYPE = long(16781312)
```

```
; The units code for millibars is 1081593921
UNITS = long(1081593921)
; We will store this as a an explicit array
STORG = long(0)

writeu,log_file_unit, RECTYPE, NDEX, START, ENDIT $
, GPTNUM, DUPNUM, DESSUP, DESFMT, DESTYPE, UNITS  $
, STORG, reserved, reserved

; set up the pressure DESCVAL record
RECTYPE = long(35)
; This belongs to Level 1, dimension number 3
LEVEL = long(1)
NDEX = long(2)
; We list the pressure levels
AVALS = float([1000., 850., 700., 500., 250., 100.] )

writeu,log_file_unit, RECTYPE, LEVEL, NDEX, AVALS

; do the time dimension spec (DESCRIP2 and DESCVAL)
writeu,log_file_unit, long(32), long(0), long(0)  $
, long(0), long(50397184), long(131072)          $
, long(1615331845), long(1), reserved, reserved
writeu,log_file_unit, long(35), long(2), long(0) $
, long(1, 1)

; set up the time averaging DESCRIP3 record
RECTYPE = long(33)
; This describes the first Level 3 dimension
NDEX = long(0)
; This covers the entire range of all Levels 0-2
; dimension
START = long( [ 0, 0, 0, 0, 0 ] )
ENDIT = long( [ -1, -1, -1, -1, -1 ] )
; There are 10 years in the average
GPTNUM = long(10)
; These are arithmetic means computed from dis-
crete sums
AVGCOD = 1
; No duplicate records
DUPNUM = long(0)
; No supplemental information
DESSUP = long(0)
; years are unsigned long integers
DESFMT = long(50397184)
```

```
; the quantity id for time is 131072
DESTYPE = long(131072)
; The units code for year is 1615331616
UNITS = long(1615331616)
; We will store this as a an implicit
; (low-value, high-value) pair
STORG = long(2)

writeu,log_file_unit, RECTYPE, NDEX, START, ENDIT $
, GPTNUM, AVGCOD, DUPNUM, DESSUP, DESFMT, DESTYPE $
, UNITS, STORG, reserved, reserved

; set up the 10-year average DESCVAL record
RECTYPE = long(35)
; This belongs to Level 3, dimension number 1
LEVEL = long(3)
NDEX = long(0)
; We start at 1983 increase and end at 1992
AVALS = long([ 1983, 1992 ] )

writeu,log_file_unit, RECTYPE, LEVEL, NDEX, AVALS
```

Note that the specifications for the dimensions, the dimension descriptors, and even the data itself do not have to be stored in any prescribed order. This is explained in Section 2.3.2. These indices are used as pointers from the descriptors to the specifications to the data itself. The DESCVAL record associated with the DESCRIP3 record will contain the year index over which the data has been averaged. These values are not needed to access the data values but instead they aid in interpreting the meaning of the data.

3.2.3 Example 3. Tensor (Wind Stress)

Next, consider a data object consisting of a horizontal wind stress tensor measured over an x-y coordinate grid at the surface of the earth. The wind stress tensor has four components, two along each of the dimensions of the grid. We take the x and y to be the coordinates of a 100 × 75 grid on which the tensors are defined. Suppose further that we attach no time to these data. (They may be time-independent estimates to be used in some sort of simple model.)

Thus, there are two Level 0 dimensions, each having 2 grid points: S_{xx}, S_{xy}, S_{yx}, S_{yy}, where S_{ij} is the wind stress in the i direction on the surface normal to the j direction. The data have two Level 1 dimensions: x and y. The data object has data records, so it must have a Level 2 dimension; we will use a generic index counter as the Level 2 dimension.

To complicate matters just a little more, we will assume that the data are written out as the array: S(wind stress in x direction, x, wind stress in y direction, y). The data object fields will be as follows (indices start at 1):

OBJDESC.	NDIM0 = 2
	NDIM1 = 2
	NDIM2 = 1
	NDIM3 = 0
DIMSPEC0.	INDEX = (1, 3)
	GPTNUM = (2, 2)
DIMSPEC1.	INDEX = (2, 4)
	DESNUM = (1, 1)
DIMSPEC2.	INDEX = -1
DESCRIP0.	DATFMT = (single precision floating-point, single precision floating-point, single precision floating-point, single precision floating-point)
	VARTYPE = (S_{xx}, S_{xy}, S_{yx}, S_{yy})
	UNITS = (Pa, Pa, Pa, Pa)
DESCRIP1.	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	START = 1
	END = -1
	GPTNUM = 100
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = x
	UNITS = no units
	STORG = 1
DESCVAL.	DLEVEL = 1
	LEVEL = 1
	DINDEX = 0
	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	AVALS = (1, 1)
DESCRIP1.	DEXSORT = 2
	NDEX = 2
	RECSORT = 0
	START = 1
	END = -1
	GPTNUM = 75
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer

	DESTYPE = y
	UNITS = no units
	STORG = 1
DESCVAL.	DLEVEL = 1
	LEVEL = 1
	DINDEX = 0
	DEXSORT = 2
	NDEX = 2
	RECSORT = 0
	AVALS = (1, 1)
DESCRIP2.	NDEX = 0
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = index
	UNITS = no units
	STORG = 1
DESCVAL.	DLEVEL = 2
	LEVEL = 2
	DINDEX = 0
	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	AVALS = (1, 1)

3.2.4 Example 4. Unusual Data Object (Ozonesondes)

Consider a data object consisting of balloon-borne observations of ozone concentrations at varying pressure levels over a set of 2000 stations on 365 days. We consider two different methods of specifying these data:

In the first case, we will define a complex data object (an "ozonesonde data vector") consisting of a single Level 0 dimension with two components: ozone concentration and pressure. The station identifier we will treat as a Level 1 dimension, and the Level 2 dimension is time. The data will be written out in the order (station ID, ozonesonde data component, time). In this case, the data object dimensional record fields will be as follows (indices start at 1):

OBJDESC.	NDIM0 = 1
	NDIM1 = 1
	NDIM2 = 1
	NDIM3 = 0
DIMSPEC0.	INDEX = 2
	GPTNUM = 2
DIMSPEC1.	INDEX = 1
	DESNUM = 1
DIMSPEC2.	INDEX = 3

	GPTNUM = 365
DESCRIP0.	DATFMT = (single precision floating-point, single precision floating-point)
	VARTYPE = (ozone concentration, pressure)
	UNITS = (ppmv, mb)
DESCRIP1.	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	START = 1
	END = -1
	GPTNUM = 2000
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = station identifier
	UNITS = no units
	STORG = 0
DESCVAL.	DLEVEL = 1
	LEVEL = 1
	DINDEX = 0
	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	AVALS = (list of 2000 station identifiers)
DESCRIP2.	NDEX = 1
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = time
	UNITS = day of year
	STORG = 1
DESCVAL.	DLEVEL = 2
	LEVEL = 2
	DINDEX = 0
	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	AVALS = (1, 1)

Alternatively, we can treat pressure as a Level 1 dimension. The first 6 months of data contain ozonesondes at the same 18 pressure levels, while the last 6 months contain the data at 15 levels, different from the set in the early period. The data array is written in the order (station identifier, pressure, time). Note that the ozone data are now simple scalars. The data dimensional record fields will be as follows (indices start at 1):

OBJDESC.	NDIM0 = 1

	NDIM1 = 2
	NDIM2 = 1
	NDIM3 = 0
DIMSPEC0.	INDEX = -1
	GPTNUM = 1
DIMSPEC1.	INDEX = (1, 2)
	DESNUM = (1, 2)
DIMSPEC2.	INDEX = 3
	GPTNUM = 365
DESCRIP0.	DATFMT = single precision floating-point
	VARTYPE = ozone concentration
	UNITS = ppmv
DESCRIP1.	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	START = 1
	END = -1
	GPTNUM = 2000
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = station identifier
	UNITS = no units
	STORG = 0
DESCVAL.	DLEVEL = 1
	LEVEL = 1
	DINDEX = 0
	DEXSORT = 1
	NDEX = 1
	RECSORT = 0
	AVALS = (list of 2000 station identifiers)
DESCRIP1.	DEXSORT = 65538
	NDEX = 2
	RECSORT = 1
	START = 1
	END = 181
	GPTNUM = 18
	DUPNUM = 0
	DESSUP = 0
	DESFMT = single precision floating-point
	DESTYPE = pressure
	UNITS = mb
	STORG = 0
DESCVAL.	DLEVEL = 1
	LEVEL = 1
	DINDEX = 0

```
             DEXSORT = 65538
                  NDEX = 2
                  RECSORT = 1
             AVALS = (list of 18 pressure levels)
DESCRIP1.    DEXSORT = 655362
                  NDEX = 2
                  RECSORT = 10
             START = 182
             END = -1
             GPTNUM = 15
             DUPNUM = 0
             DESSUP = 0
             DESFMT = single precision floating-point
             DESTYPE = pressure
             UNITS = mb
             STORG = 0
DESCVAL.     DLEVEL = 1
                  LEVEL = 1
                  DINDEX = 0
             DEXSORT = 655362
                  NDEX = 2
                  RECSORT = 10
             AVALS = (list of 15 pressure levels)
DESCRIP2.    NDEX = 1
             DUPNUM = 0
             DESSUP = 0
             DESFMT = long integer
             DESTYPE = time
             UNITS = day of year
             STORG = 1
DESCVAL.     DLEVEL = 2
                  LEVEL = 2
                  DINDEX = 0
             DEXSORT = 1
                  NDEX = 1
                  RECSORT = 0
             AVALS = (1, 1)
```

3.2.5 Example 5. Nonauthoritative Dimension Descriptors (Ozonesondes)

This example is an extension of the previous example and is a more realistic specification of ozonesondes. This example looks at a very small set of data, but the data varies over all dimensions. Again, consider a data object consisting of balloon-borne observations of ozone concentrations at varying pressure levels, this time over a set of 4 stations (1001, 1002, 1003, 1004) on 3 days (21-23 August 1992). Our Level 0 dimension will consist of ozone concentration. The Level 1 dimension will be the pressure levels at each station where the observations are recorded. There will be one Level 2 dimension consisting of a generic index. The station identifiers and the date (specified as yymmdd, where yy = last two digits of year, mm = month number, and dd = day of month) will be additional descriptions for the Level 2 dimension. This is the general way to specify multiple varying Level 2 dimensions.

The observations are taken as follows: On 21 August 1992, station 1001 reported observations at 15 pressure levels, station 1002 at 19 pressure levels, station 1003 at 18 pressure levels, and station 1004 did not report. On 22 August 1992, station 1001 reported observations at 6 pressure levels, station 1002 at 11 pressure levels, station 1003 at 20 pressure levels, and station 1004 at 22 pressure levels. On 23 August 1992, station 1001 reported observations at 12 pressure levels, station 1002 at 17 pressure levels, station 1003 did not report, and station 1004 at 13 pressure levels. In this case, the data object dimensional record fields will be as follows (indices start at 0):

```
OBJDESC.      NDIM0 = 1
              NDIM1 = 1
              NDIM2 = 1
              NDIM3 = 0
DIMSPEC0.     INDEX = -1
              GPTNUM = 1
DIMSPEC1.     INDEX = 0
              DESNUM = 10
DIMSPEC2.     INDEX = 1
              GPTNUM = 10
DESCRIP0.     DATFMT = single precision floating-point
              VARTYPE = ozone concentration
              UNITS = ppmv
DESCRIP1.     DEXSORT = 0
                  NDEX = 0
                  RECSORT = 0
              START = 0
              END = 0
              GPTNUM = 15
              DUPNUM = 0
              DESSUP = 0
              DESFMT = single precision floating-point
              DESTYPE = pressure
              UNITS = mb
              STORG = 0
DESCVAL.      DLEVEL = 1
                  LEVEL = 1
                  DINDEX = 0
              DEXSORT = 0
                  NDEX = 0
```

RECSORT = 0
AVALS = (list of 15 pressure levels)
DESCRIP1. DEXSORT = 65536
NDEX = 0
RECSORT = 1
START = 1
END = 1
GPTNUM = 19
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 65536
NDEX = 0
RECSORT = 1
AVALS = (list of 19 pressure levels)
DESCRIP1. DEXSORT = 131072
NDEX = 0
RECSORT = 2
START = 2
END = 2
GPTNUM = 18
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 131072
NDEX = 0
RECSORT = 2
AVALS = (list of 18 pressure levels)
DESCRIP1. DEXSORT = 196608
NDEX = 0
RECSORT = 3
START = 3
END = 3
GPTNUM = 6

```
             DUPNUM = 0
             DESSUP = 0
             DESFMT = single precision floating-point
             DESTYPE = pressure
             UNITS = mb
             STORG = 0
DESCVAL.     DLEVEL = 1
                 LEVEL = 1
                 DINDEX = 0
             DEXSORT = 196608
                 NDEX = 0
                 RECSORT = 3
             AVALS = (list of 6 pressure levels)
DESCRIP1.    DEXSORT = 262144
                 NDEX = 0
                 RECSORT = 4
             START = 4
             END = 4
             GPTNUM = 11
             DUPNUM = 0
             DESSUP = 0
             DESFMT = single precision floating-point
             DESTYPE = pressure
             UNITS = mb
             STORG = 0
DESCVAL.     DLEVEL = 1
                 LEVEL = 1
                 DINDEX = 0
             DEXSORT = 262144
                 NDEX = 0
                 RECSORT = 4
             AVALS = (list of 11 pressure levels)
DESCRIP1.    DEXSORT = 327680
                 NDEX = 0
                 RECSORT = 5
             START = 5
             END = 5
             GPTNUM = 20
             DUPNUM = 0
             DESSUP = 0
             DESFMT = single precision floating-point
             DESTYPE = pressure
             UNITS = mb
             STORG = 0
DESCVAL.     DLEVEL = 1
                 LEVEL = 1
```

DINDEX = 0
DEXSORT = 327680
NDEX = 0
RECSORT = 5
AVALS = (list of 20 pressure levels)
DESCRIP1. DEXSORT = 393216
NDEX = 0
RECSORT = 6
START = 6
END = 6
GPTNUM = 22
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 393216
NDEX = 0
RECSORT = 6
AVALS = (list of 22 pressure levels)
DESCRIP1. DEXSORT = 458752
NDEX = 0
RECSORT = 7
START = 7
END = 7
GPTNUM = 12
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 458752
NDEX = 0
RECSORT = 7
AVALS = (list of 12 pressure levels)
DESCRIP1. DEXSORT = 524288
NDEX = 0
RECSORT = 8

START = 8
END = 8
GPTNUM = 17
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 524288
NDEX = 0
RECSORT = 8
AVALS = (list of 17 pressure levels)
DESCRIP1. DEXSORT = 589824
NDEX = 0
RECSORT = 9
START = 9
END = 9
GPTNUM = 13
DUPNUM = 0
DESSUP = 0
DESFMT = single precision floating-point
DESTYPE = pressure
UNITS = mb
STORG = 0
DESCVAL. DLEVEL = 1
LEVEL = 1
DINDEX = 0
DEXSORT = 589824
NDEX = 0
RECSORT = 9
AVALS = (list of 13 pressure levels)
DESCRIP2. NDEX = 0
DUPNUM = 2
DESSUP = 0
DESFMT = long integer
DESTYPE = generic index
UNITS = no units
STORG = 1
DESCVAL. DLEVEL = 2
LEVEL = 2
DINDEX = 0
DEXSORT = 0

	NDEX = 0
	RECSORT = 0
	AVALS = (0, 1)
DESCRIP.	LEVEL = 2
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = station identifier
	UNITS = no units
	STORG = 0
	DINDEX = 1
DESCVAL.	DLEVEL = 6
	LEVEL = 2
	DINDEX = 1
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	AVALS = (1001, 1002, 1003, 1001, 1002, 1003, 1004, 1001, 1002, 1004)
DESCRIP.	LEVEL = 2
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = time
	UNITS = YYMMDD
	STORG = 0
	DINDEX = 2
DESCVAL.	DLEVEL = 10
	LEVEL = 2
	DINDEX = 2
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	AVALS = (920821, 920821, 920821, 920822, 920822, 920822, 920822, 920823, 920823, 920823)

Note that the START and END fields of DESCRIP1 contain the indices for the days and stations over which the different pressure level definitions apply. This is explained further in Section 2.3.2. The DESCVAL record associated with each DESCRIP1 record contains the appropriate values of pressure.

These START and END fields lead us into the next set of examples.

3.3 START and END Fields

START and END specify the ranges of data dimensions over which certain records apply. In most cases, the records will apply over the full range of all dimensions, so that the values for START will be 1 (or 0, depending on the beginning index value specified in TEST.SPECB.IDXSTART) and -1 for END. (The -1 value indicates the last grid point value, and so the two together will indicate the whole range of a dimension.)

3.3.1 DESCRIP1 Case

The START and END fields in the DESCRIP1 record type indicate which Level 2 dimensions the DESCRIP1 record applies to. Each data record (Level 2 dimension) may have a different number of Level 1 dimensions contained within it. The DESCRIP1.GPTNUM field indicates the number of grid points in that Level 1 dimension.

In the second part of Example 4 from Section 3.2.4, we specified two separate descriptors for the pressure dimension. The first half of the year had 18 pressure levels, and the second half of the year had a different set of 15 pressure levels. Two DESCRIP1 records were used, then, to describe the pressure dimension. The first record had START = 1 and END = 181 (for 1 January to 30 June); the second record had START = 182 and END = 365 (for 1 July to 31 December).

Consider another example in a bit more detail. We have an array of station identifiers which contain observations of surface temperatures with time. Not all stations report every time observation; we store only those that do. Suppose, for example, that we have observations for stations 1001, 1002, and 1003 on days 1 and 3, and stations 1001 and 1002 on day 2. Assume that all of the temperatures for a particular time are to be written in one record.

Temperature is a scalar, so there is a single Level 0 dimension with one grid point. We consider the station identifier to be a Level 1 dimension. Time is a Level 2 dimension, and there are 31 days of data.

The data are written out in the order (station identifier, day). Because the station IDs change for each of the three days (i.e., for each grid point along the time dimension), three DESCRIP1 records will be needed.

The data dimension record fields will be as follows (indices start at 1):

OBJDESC.	NDIM0 = 1
	NDIM1 = 1
	NDIM2 = 1
	NDIM3 = 0
DIMSPEC0.	INDEX = -1
	GPTNUM = 1
DIMSPEC1.	INDEX = 1
	DESNUM = 3
DIMSPEC2.	INDEX = 2
	GPTNUM = 3
DESCRIP0.	DATFMT = single precision floating-point

```
           VARTYPE = surface temperature
           UNITS = K
DESCRIP1.  DEXSORT = 1
               NDEX = 1
               RECSORT = 0
           START = 1
           END = 1
           GPTNUM = 3
           DUPNUM = 0
           DESSUP = 0
           DESFMT = long integer
           DESTYPE = station identifier
           UNITS = no units
           STORG = 0
DESCVAL.   DLEVEL = 1
               LEVEL = 1
               DINDEX = 0
           DEXSORT = 1
               NDEX = 1
               RECSORT = 0
           AVALS = (1001, 1002, 1003)
DESCRIP1.  DEXSORT = 65537
               NDEX = 1
               RECSORT = 1
           START = 2
           END = 2
           GPTNUM = 2
           DUPNUM = 0
           DESSUP = 0
           DESFMT = long integer
           DESTYPE = station identifier
           UNITS = no units
           STORG = 0
DESCVAL.   DLEVEL = 1
               LEVEL = 1
               DINDEX = 0
           DEXSORT = 65537
               NDEX = 1
               RECSORT = 1
           AVALS = (1001, 1002)
DESCRIP1.  DEXSORT = 131073
               NDEX = 1
               RECSORT = 2
           START = 3
           END = 3
           GPTNUM = 3
```

```
            DUPNUM = 0
            DESSUP = 0
            DESFMT = long integer
            DESTYPE = station identifier
            UNITS = no units
            STORG = 0
DESCVAL.    DLEVEL = 1
                LEVEL = 1
                DINDEX = 0
            DEXSORT = 131073
                NDEX = 1
                RECSORT = 2
            AVALS = (1001, 1002, 1003)
DESCRIP2.   NDEX = 1
            DUPNUM = 0
            DESSUP = 0
            DESFMT = long integer
            DESTYPE = time
            UNITS = day
            STORG = 1
DESCVAL.    DLEVEL = 2
                LEVEL = 2
                DINDEX = 0
            DEXSORT = 1
                NDEX = 1
                RECSORT = 0
            AVALS = (1, 1)
```

3.3.2 BADVAL Case

The BADVAL START and END fields will specify which Level 1 and Level 2 dimensions a given BADVAL record will apply to. Again, there can be no overlap of the ranges specified by multiple BADVAL records; i.e., there can only be one bad-data flag per data value.

In the previous example, we changed the number of station identifiers in each record, depending on which stations had data available. Another option would be to make a single Level 1 definition for the array of station identifiers, giving it a fixed length of three. In this case, we would mark the data from Station 1003 on Day 2 with bad-data flag values. That is, we would set the missing station's temperature to some flag value and set the BADVAL.VALUE field to that same flag value. The START and END fields have two elements each (one for the Level 1 dimension and one for the Level 2 dimension); to make this BADVAL record apply over the entire dataset, both START elements would be 0, and both END elements would be -1. This is the most straightforward and common use of the BADVAL record type. The relevant record fields will be as follows (indices start at 0):

OBJDESC.	NDIM0 = 1
	NDIM1 = 1
	NDIM2 = 1
	NDIM3 = 0
	NBADS = 1
DIMSPEC0.	INDEX = -1
	GPTNUM = 1
DIMSPEC1.	INDEX = 0
	DESNUM = 1
DIMSPEC2.	INDEX = 1
	GPTNUM = 3
DESCRIP0.	DATFMT = single precision floating-point
	VARTYPE = temperature
	UNITS = K
DESCRIP1.	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	START = 0
	END = -1 or 2
	GPTNUM = 3
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = station identifier
	UNITS = no units
	STORG = 0
DESCVAL.	DLEVEL = 1
	LEVEL = 1
	DINDEX = 0
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0
	AVALS = (1001, 1002, 1003)
DESCRIP2.	NDEX = 0
	DUPNUM = 0
	DESSUP = 0
	DESFMT = long integer
	DESTYPE = time
	UNITS = day
	STORG = 1
DESCVAL.	DLEVEL = 2
	LEVEL = 2
	DINDEX = 0
	DEXSORT = 0
	NDEX = 0
	RECSORT = 0

	AVALS = (1, 1)
BADVAL.	RECSORT = 0
	START = (0, 0)
	END = (-1, -1)
	VALUE = bad-data flag

As another example, consider the case of the vector winds described above in Example 2 in Section 3.2.2. In this case, a different bad-data flag may be needed for each data component. The elements of the START and END array will correspond to the longitude, latitude, pressure, and day-of-year dimensions. The relevant record fields will be the same as in that example, with the addition of the BADVAL record type as follows (indices start at 0):

OBJDESC.	NBADS = 1
BADVAL.	RECSORT = 0
	START = (0, 0, 0, 0)
	END = (-1, -1, -1, -1)
	VALUE = u bad-data flag, v bad-data flag, w bad-data flag

The IDL code for writing such a record might look something like this:

```
    ; set up the BADVAL record
    RECTYPE = long(40)
    ; This is the first BADVAL record
    RECSORT = long(1)
    ; This BADVAL record is valid over the en-
tire range of
    ; Levels 1-3 dimensions
    START = long( [ 0, 0, 0, 0 ] )
    ENDIT = long( [ -1, -1, -1, -1 ])
    ; Remember, VALUE must have as many compo-
nents as the data
    VALUE = float( [ -9999.0, -9999.0, -999.0 ] )

    writeu,log_file_unit,  RECTYPE,RECSORT, reserved $
    , reserved, START, ENDIT, VALUE
```

Now consider the possibility that after, say, nine days the bad-data flag is changed, so that each pressure altitude has its own bad-data flag value. This situation would need seven BADVAL records: the first would contain the single bad data value that applies to all of the pressure altitudes for the first nine days. Then a second BADVAL record would contain the flag that applies to the first pressure altitude for the rest of the days. The third through seventh records would give the flags that apply to the other pressure levels after the ninth day.

The relevant record fields would now be as follows (indices start at 0):

OBJDESC.	NBADS = 7
BADVAL.	RECSORT = 0
	START = (0, 0, 0, 0)
	END = (-1, -1, -1, 8)
	VALUE= first bad data value
BADVAL.	RECSORT = 1
	START = (0, 0, 0, 9)
	END = (-1, -1, 0, -1)
	VALUE= second bad data value at pressure level 0
BADVAL.	RECSORT = 2
	START = (0, 0, 1, 9)
	END = (-1, -1, 1, -1)
	VALUE= second bad data value at pressure level 1
BADVAL.	RECSORT = 3
	START = (0, 0, 2, 9)
	END = (-1, -1, 2, -1)
	VALUE= second bad data value at pressure level 2
BADVAL.	RECSORT = 4
	START = (0, 0, 3, 9)
	END = (-1, -1, 3, -1)
	VALUE= second bad data value at pressure level 3
BADVAL.	RECSORT = 5
	START = (0, 0, 4, 9)
	END = (-1, -1, 4, -1)
	VALUE = second bad data value at pressure level 4
BADVAL.	RECSORT = 6
	START = (0, 0, 5, 9)
	END = (-1, -1, 5, -1)
	VALUE = second bad data value at pressure level 5

3.3.3 DESCRIP3, PROCSPEC, PROCDUP, AUXSPEC, and PAKSPEC Cases

These record types use the START and END fields in the same manner, so they are dealt with together here. The only exception is that the fields in the DESCRIP3 records for the same Level 3 dimension must not overlap, while there can be any overlap in the PROCSPEC, PROCDUP, AUXSPEC, and PAKSPEC record types.

We will consider again the wind vector example from Example 2 in Section 3.2.2. As described, the START and END fields of the DESCRIP3 record extend over all dimensions of the data object. This is the most straightforward use of these fields.

Now suppose that we process the Northern Hemisphere differently from the Southern Hemisphere, and furthermore, we change the southern hemisphere processing from day 20 onwards. These three types of processing would be compactly represented as locally defined integer codes, which we denote as “codenh,” “codesh1,” and “codesh2.” We would need, then, three PROCSPEC records to define these processing codes. Note that the data at the equator, because they may be considered to belong to both hemispheres, will be associated with two processing codes: one for the northern hemisphere and one for the southern. The elements of the START and END arrays will correspond to the wind component, longitude, latitude, pressure, and day-of-year dimensions. The relevant record fields will contain the following (indices start at 1):

OBJDESC.	NPROCS = 3
PROCSPEC.	RECSORT = 1
	START = (1, 1, 46, 1, 1)
	END = (-1, -1, 91, -1, -1)
	CODE = codenh
	PRCNUM = 1
	NDUPS = 0
PROCFORM.	RECSORT = 1
	PRCFMT = long integer
PROCVAL.	RECSORT = 1
	INFO = codenh information
PROCSPEC.	RECSORT = 2
	START = (1, 1, 1, 1, 1)
	END = (-1, -1, 46, -1, 19)
	CODE = codesh1
	PRCNUM = 1
	NDUPS = 0
PROCFORM.	RECSORT = 2
	PRCFMT = long integer
PROCVAL.	RECSORT = 2
	INFO = codesh1 information
PROCSPEC.	RECSORT = 3
	START = (1, 1, 1, 1, 20)
	END = (-1, -1, 46, -1, -1)
	CODE = codesh2
	PRCNUM = 1
	NDUPS = 0
PROCFORM.	RECSORT = 3
	PRCFMT = long integer
PROCVAL.	RECSORT = 3
	INFO = codesh2 information

3.4 Audit Trail

We will consider an example where there are three sites, numbered 501, 502, and 503. On 20 January 1989 at Site 501, a program with a Task number 1234 generates a data set which is then reprocessed by Task 2452 on 6 October 1990. Meanwhile, at Site 502, a Task number 1234 (which almost certainly denotes a different task than Site 501's Task 1234) also generates a dataset on 7 October 1990. Finally, Site 503 receives the datasets from Site 501 and Site 502 and uses Task number 8364 to incorporate these data into calculations that produce a new dataset on 15 July 1991. We will trace through these datasets using the notation (site code, task code, date, pointer).

The first data sets created at Site 501 and Site 502 had no previous history and therefore contain one node which contains information about the current dataset. The AUDIT.TREE field will contain:

Site 1, dataset 1: (501, 1234, 32528, 0)
Site 2, dataset 1: (502, 1234, 33153, 0)

Note that the 20 January 1989 is 32,528 days after 31 December 1899, while 33,153 days after is 7 October 1990. Also, the pointer fields are set to 0 to indicate that there are no further nodes in the tree.

When Site 501 reprocesses its first dataset, the current audit node will be appended to the audit node from that dataset, and the first audit node will change its pointer from 0 to 1. The new AUDIT.TREE field will contain:

Site 1, dataset:2 (501, 1234, 32528, 1) (501, 2542, 33152, 0)

Finally, when Site 3 takes these two datasets and creates another one, the two AUDIT.TREE nodes from the two datasets are concatenated with a new node for the generated dataset. The NULL pointers will change to point to the newly added node. The final AUDIT.TREE field will contain:

(501, 1234, 32528, 1)
(501, 2542, 33152, 2)
(5032, 1234, 33153, 1)
(503, 8364, 33434, 0)

We can define an algorithm to create new nodes and append them to other nodes as follows:

1. For NUM = the number of component datasets used in creating the current dataset
   $N(i) =$ the number of nodes in the i component data set's audit tree (i = 1,NUM)
2. If NUM = 0 then skip to step 6
3. For every i, where i = 1,NUM do steps 4 and 5
4. Set NODE($N(i)$).POINTER = $\mathrm{sum}(N(j)) + 1)$, where $j = i + 1$,NUM
5. Append NODE to the new AUDIT.TREE field
6. Create a new node, NN:
   NN.SITE = site code
   NN.TASK = task code
   NN.DATE = date (days since 1 Jan 1900)
   NN.POINTER = NIL (0)
7. Append NN to the new AUDIT.TREE field

3.5 Auxiliary Information

Following are examples of specifications for auxiliary information. All of the examples will be based on the wind vector example from Section 3.2.2.

3.5.1 Example 1. Referring to a Single Dimension, Applied to a Single Dimension

Consider that the pressure levels in the wind vector example have measured uncertainties of $[1000\pm7, 850\pm5, 700\pm5, 500\pm2, 250\pm1, 100\pm1]mb$ for all wind components, horizontal locations, and times. If the indices start numbering from 0, then we have the following auxiliary record fields:

OBJDESC.	NAUX = 1
AUXSPEC.	RECSORT = 0
	NUMREF = 1
	NUMAPP = 1
	START = (0, 0, 0, 0, 0)
	END = (-1, -1, -1, -1, -1)
	AUXFMT = single precision floating-point
	AUXTYPE = measured uncertainty
	UNITS = mb
	NUMSUP = 0
AUXRANGE.	RECSORT = 0
	REFLEV = 1
	REFNDEX = 2
	APPLEV = 1
	APPNDEX = 2
AUXVAL.	RECSORT = 0
	AVALS = (7., 5., 5., 2., 1., 1.)

In IDL, then, code to define the auxiliary information for this data object would look like this (using the basic structure as already given in Section 3.2.2):

```
    ; set up the relevant elements in the OBJDESC record
    RECTYPE = long(1)
    NDIM0 = long(1)
    NDIM1 = long(2)
    NDIM2 = long(2)
    NDIM3 = long(0)
    NAUX = long(1)
    .
    .
    .
    writeu,log_file_unit,   RECTYPE, vartype, re-
served    $
```

```
    , NDIM0, NDIM1, NDIM2, NDIM3, re-
served, isource       $
    , reserved, reserved, naudit, ninfo, ncomms, com-
cod   $
    , reserved, nbads, reserved, nprocs, re-
served, npacks $
    , reserved, NAUX, cmpnum, reserved, re-
served, reserved

    ; set up the dimensional information, bad data val-
ues,
    ; packing information, and processing information

    .
    .
    .

    ; set up the relevant elements for the AUXSPEC record
    RECTYPE = long(80)
    ; only one record group
    RECSORT = long(0)
    ; there is only one dimension with uncertainties:
    ; pressure
    NUMREF = long(1)
    ; the uncertainties vary only with pressure, there-
fore,
    ; AUXVAL.AVALS has one dimension
    NUMAPP = long(1)
    ; this covers the entire range of the data
    START = long( [0, 0, 0, 0, 0] )
    ENDIT = long( [-1, -1, -1, -1, -1] )
    ; the uncertainties are floating-point numbers
    AUXFMT = long(67108864)
    ; the quantities are measured uncertainties
    AUXTYPE = long(262144)
    ; the units are in millibars
    UNITS = long(1081593921)
    ; there are no supplemental values
    NUMSUP = long(0)

    write,log_file_unit, RECTYPE, RECSORT, NUM-
REF, NUMAPP $
    , START, ENDIT, AUXFMT, AUXTYPE, UNITS, NUMSUP $
    , reserved, reserved
```

```
    ; set up the indices in the AUXRANGE record
    RECTYPE = long(81)
    ; only one record group
    RECSORT = long(0)
    ; the uncertainties refer to pres-
sure (the third index
    ; in the Level 1 dimensions)
    REFLEV = long(1)
    REFNDEX = long(2)
    ; the uncertainties only vary over pressure (the
    ; third index in the Level 1 dimensions)
    APPLEV = long(1)
    APPNDEX = long(2)

    write,log_file_unit, RECTYPE, REC-
SORT, REFLEV, REFNDEX $
    , APPLEV, APPNDEX, reserved, reserved

    ; set up the auxiliary information in the AUXVAL record
    RECTYPE = long(82)
    ; only one record group
    RECSORT = long(0)
    AVALS = (7., 5., 5., 2., 1., 1.)

    write,log_file_unit, RECTYPE, RECSORT, AVALS
```

3.5.2 Example 2. Referring to a Single Dimension, Applied to Multiple Dimensions

Reconsider the previous example, where the measured uncertainties in the pressure levels now vary over spatial location and time, but are the same for each wind component. The uncertainties are specified in an array, U, with the dimensions of $6 \times 91 \times 72 \times 31$ (i.e., pressure, latitude, longitude, day of month). If the indices start numbering from 0, then we have the following auxiliary record fields:

OBJDESC.	NAUX = 1
AUXSPEC.	RECSORT = 0
	NUMREF = 1
	NUMAPP = 4
	START = (0, 0, 0, 0, 0)
	END = (-1, -1, -1, -1, -1)
	AUXFMT = single precision floating-point
	AUXTYPE = measured uncertainty
	UNITS = mb
	NUMSUP = 0

AUXRANGE.	RECSORT = 0
	REFLEV = 1
	REFNDEX = 2
	APPLEV = (1, 1, 1, 2)
	APPNDEX = (3, 2, 1, 0)
AUXVAL.	RECSORT = 0
	AVALS = U(6, 91, 72, 31)

3.5.3 Example 3. Referring to a Single Dimension, Applied to Multiple Dimensions and Subsections of the Data

Reconsider the previous examples, where now there are different measured uncertainties in the pressure at each pressure level. The uncertainties are different for each day in the first 20 days and then are the same each day for the last 11 days. There are also measured uncertainties in the vertical component of the wind vector that vary over latitude, longitudes, and pressure levels, but are the same for each day.

The uncertainties in the pressures for the first 20 days are specified in an array, U1, with the dimensions of 6×20 (i.e., pressure, day of month). The uncertainties in the pressures for the last 11 days are specified in an array, U2, containing 6 values (uncertainties over pressure). The uncertainties in the vertical wind field are specified in an array, U3, with the dimensions of $6 \times 91 \times 72$ (i.e., pressure, latitude, longitude). If the indices start numbering from 0, then we have the following auxiliary record fields:

OBJDESC.	NAUX = 3
AUXSPEC.	RECSORT = 0
	NUMREF = 1
	NUMAPP = 2
	START = (0, 0, 0, 0, 0)
	END = (-1, -1, -1, -1, 19)
	AUXFMT = single precision floating-point
	AUXTYPE = measured uncertainty
	UNITS = mb
	NUMSUP = 0
AUXRANGE.	RECSORT = 0
	REFLEV = 1
	REFNDEX = 2
	APPLEV = (1, 2)
	APPNDEX = (2, 0)
AUXVAL.	RECSORT = 0
	AVALS = U1(6, 20)
AUXSPEC.	RECSORT = 10
	NUMREF = 1
	NUMAPP = 1
	START = (0, 0, 0, 0, 20)
	END = (-1, -1, -1, -1, -1)
	AUXFMT = single precision floating-point

	AUXTYPE = measured uncertainty
	UNITS = mb
	NUMSUP = 0
AUXRANGE.	RECSORT = 10
	REFLEV = 1
	REFNDEX = 2
	APPLEV = 1
	APPNDEX = 2
AUXVAL.	RECSORT = 10
	AVALS = U2(6)
AUXSPEC.	RECSORT = 20
	NUMREF = 1
	NUMAPP = 3
	START = (2, 0, 0, 0, 0)
	END = (2, -1, -1, -1, -1)
	AUXFMT = single precision floating-point
	AUXTYPE = measured uncertainty
	UNITS = cm/s
	NUMSUP = 0
AUXRANGE.	RECSORT = 20
	REFLEV = 0
	REFNDEX = 2
	APPLEV = (1, 1, 1)
	APPNDEX = (2, 1, 0)
AUXVAL.	RECSORT = 20
	AVALS = U3(6, 91, 72)

In IDL, then, code to define the auxiliary information for this data object would look like this (using the basic structure as already given in Section 3.2.2):

```
    ; set up the relevant elements in the OBJDESC record
    RECTYPE = long(1)
    NDIM0 = long(1)
    NDIM1 = long(2)
    NDIM2 = long(2)
    NDIM3 = long(0)
    NAUX = long(3)
    .
    .
    .
    writeu,log_file_unit,   RECTYPE, vartype, re-
served    $
    , NDIM0, NDIM1, NDIM2, NDIM3, re-
served, isource       $
    , reserved, reserved, naudit, ninfo, ncomms, com-
cod   $
```

```
    , reserved, nbads, reserved, nprocs, re-
served, npacks $
    , reserved, NAUX, cmpnum, reserved, re-
served, reserved

    ; set up the dimensional information, bad data val-
ues,
    ; packing information, and processing information

    .
    .
    .

    ; set up the relevant elements for the first AUXSPEC
    ; record: pressure uncertainties for the first 20 days
    RECTYPE = long(80)
    ; first of three record groups
    RECSORT = long(0)
    ; there is only one dimension with uncertainties:
    ; pressure
    NUMREF = long(1)
    ; the uncertainties vary with pressure and time,
    ; therefore, AUXVAL.AVALS has two dimensions
    NUMAPP = long(2)
    ; this covers the entire spatial range of the data and all
    ; of the data components for the first 20 days
    START = long( [0, 0, 0, 0, 0] )
    ENDIT = long( [-1, -1, -1, -1, 19] )
    ; the uncertainties are floating-point numbers
    AUXFMT = long(67108864)
    ; the quantities are measured uncertainties
    AUXTYPE = long(262144)
    ; the units are in millibars
    UNITS = long(1081593921)
    ; there are no supplemental values
    NUMSUP = long(0)

    write,log_file_unit, RECTYPE, RECSORT, NUM-
REF, NUMAPP $
    , START, ENDIT, AUXFMT, AUXTYPE, UNITS, NUMSUP $
    , reserved, reserved

    ; set up the indices in the AUXRANGE record
    RECTYPE = long(81)
    ; first of three record groups
```

```
    RECSORT = long(0)
    ; the uncertainties refer to pres-
sure (the third index
    ; in the Level 1 dimensions
    REFLEV = long(1)
    REFNDEX = long(2)
    ; the uncertainties vary over pres-
sure (the third index
    ; in the Level 1 dimensions) and time (the first index
    ; in the Level 2 dimensions)
    APPLEV = long( [1, 2] )
    APPNDEX = long( [2, 0] )

    write,log_file_unit, RECTYPE, REC-
SORT, REFLEV, REFNDEX $
    , APPLEV, APPNDEX, reserved, reserved

    ; set up the auxiliary information in the AUXVAL record
    RECTYPE = long(82)
    ; first of three record groups
    RECSORT = long(0)
    AVALS = U1

    write,log_file_unit, RECTYPE, RECSORT, AVALS

    ; set up the relevant elements for the second AUXSPEC
    ; record: pressure uncertainties for the last 11 days
    RECTYPE = long(80)
    ; second of three record groups
    RECSORT = long(10)
    ; there is only one dimension with uncertainties:
    ; pressure
    NUMREF = long(1)
    ; the uncertainties vary only with pressure, there-
fore,
    ; AUXVAL.AVALS has one dimension
    NUMAPP = long(1)
    ; this covers the entire spatial range of the data and all
    ; of the data components for the last 11 days
    START = long( [0, 0, 0, 0, 20] )
    ENDIT = long( [-1, -1, -1, -1, -1] )
    ; the uncertainties are floating-point numbers
    AUXFMT = long(67108864)
    ; the quantities are measured uncertainties
    AUXTYPE = long(262144)
```

```
    ; the units are in millibars
    UNITS = long(1081593921)
    ; there are no supplemental values
    NUMSUP = long(0)

    write,log_file_unit, RECTYPE, RECSORT, NUM-
REF, NUMAPP $
    , START, ENDIT, AUXFMT, AUXTYPE, UNITS, NUMSUP $
    , reserved, reserved

    ; set up the indices in the AUXRANGE record
    RECTYPE = long(81)
    ; second of three record groups
    RECSORT = long(10)
    ; the uncertainties refer to pressure (the third
    ; the third index in the Level 1 dimensions
    REFLEV = long(1)
    REFNDEX = long(2)
    ; the uncertainties vary over pres-
sure (the third index
    ; in the Level 1 dimensions)
    APPLEV = long(1)
    APPNDEX = long(2)

    write,log_file_unit, RECTYPE, REC-
SORT, REFLEV, REFNDEX $
    , APPLEV, APPNDEX, reserved, reserved

    ; set up the auxiliary information in the AUXVAL record
    RECTYPE = long(82)
    ; second of three record groups
    RECSORT = long(10)
    AVALS = U2

    write,log_file_unit, RECTYPE, RECSORT, AVALS

    ; set up the relevant elements for the third AUXSPEC
    ; record: vertical wind uncertainties
    RECTYPE = long(80)
    ; third of three record groups
    RECSORT = long(20)
    ; there is only one dimension with uncertainties:
    ; vertical wind component
    NUMREF = long(1)
    ; the uncertainties vary with pressure, latitude, and
```

```
    ; longitude, therefore, AUXVAL.AVALS has three dimensions
    NUMAPP = long(3)
    ; this covers the entire spatial range of the data and all
    ; days for the vertical wind component
    START = long( [2, 0, 0, 0, 0] )
    ENDIT = long( [2, -1, -1, -1, -1] )
    ; the uncertainties are floating-point numbers
    AUXFMT = long(67108864)
    ; the quantities are measured uncertainties
    AUXTYPE = long(262144)
    ; the units are in cm/s
    UNITS = long(1616347137)
    ; there are no supplemental values
    NUMSUP = long(0)

    write,log_file_unit, RECTYPE, RECSORT, NUM-
REF, NUMAPP $
    , START, ENDIT, AUXFMT, AUXTYPE, UNITS, NUMSUP $
    , reserved, reserved

    ; set up the indices in the AUXRANGE record
    RECTYPE = long(81)
    ; third of three record groups
    RECSORT = long(20)
    ; the uncertainties refer to vertical wind, which is
    ; the third index in the Level 0 dimensions
    REFLEV = long(0)
    REFNDEX = long(2)
    ; the uncertainties vary over pres-
sure (the third index
    ; in the Level 1 dimensions), latitude (the sec-
ond index
    ; in the Level 1 dimensions), and longi-
tude (the first
    ; index in the Level 1 dimensions)
    APPLEV = long( [1, 1, 1] )
    APPNDEX = long( [2, 1, 0] )

    write,log_file_unit, RECTYPE, REC-
SORT, REFLEV, REFNDEX $
    , APPLEV, APPNDEX, reserved, reserved

    ; set up the auxiliary information in the AUXVAL record
    RECTYPE = long(82)
    ; third of three record groups
    RECSORT = long(20)
    AVALS = U3
```

```
write,log_file_unit, RECTYPE, RECSORT, AVALS
```

3.5.4 Example 4. Referring to Multiple Dimensions, Applied to Multiple Dimensions and Subsections of the Data

Again, consider the previous examples, where now there are different measured uncertainties associated with the longitudes and latitudes, where the uncertainties vary over each longitude band in the southern hemisphere (including the equator). In the northern hemisphere, the measured uncertainties vary with each longitude–latitude grid point.

The uncertainties in the southern hemisphere are specified in an array, U1, containing 72 values (uncertainties over longitude). The uncertainties in the northern hemisphere are specified in an array, U2, with the dimensions of 45×72 (i.e., latitude, longitude). If the indices start numbering from 0, then we have the following auxiliary record fields:

OBJDESC.	NAUX = 2
AUXSPEC.	RECSORT = 0
	NUMREF = 2
	NUMAPP = 1
	START = (0, 0, 0, 0, 0)
	END = (-1, -1, 45, -1, 0)
	AUXFMT = single precision floating-point
	AUXTYPE = measured uncertainty
	UNITS = deg
	NUMSUP = 0
AUXRANGE.	RECSORT = 0
	REFLEV = (1, 1)
	REFNDEX = (0, 1)
	APPLEV = 1
	APPNDEX = 0
AUXVAL.	RECSORT = 0
	AVALS = U1(72)
AUXSPEC.	RECSORT = 10
	NUMREF = 2
	NUMAPP = 2
	START = (0, 0, 46, 0, 0)
	END = (-1, -1, -1, -1, -1)
	AUXFMT = single precision floating-point
	AUXTYPE = uncertainty
	UNITS = deg
	NUMSUP = 0
AUXRANGE.	RECSORT = 10
	REFLEV = (1, 1)
	REFNDEX = (0, 1)
	APPLEV = (1, 1)
	APPNDEX = (1, 0)

AUXVAL. RECSORT = 10
AVALS = U2(45,72)

In IDL, then, code to define the auxiliary information for this data object would look like this (using the basic structure as already given in Section 3.2.2):

```
    ; set up the relevant elements in the OBJDESC record
    RECTYPE = long(1)
    NDIM0 = long(1)
    NDIM1 = long(2)
    NDIM2 = long(2)
    NDIM3 = long(0)
    NAUX = long(2)
    .
    .
    .
    writeu,log_file_unit,   RECTYPE, vartype, re-
served    $
    , NDIM0, NDIM1, NDIM2, NDIM3, re-
served, isource       $
    , reserved, reserved, naudit, ninfo, ncomms, com-
cod   $
    , reserved, nbads, reserved, nprocs, re-
served, npacks $
    , reserved, NAUX, cmpnum, reserved, re-
served, reserved


    ; set up the dimensional information, bad data val-
ues,
    ; packing information, and processing information

    .
    .
    .

    ; set up the relevant elements for the first AUXSPEC
    ; record: longitude and latitude uncertainties in the
    ; southern hemisphere
    RECTYPE = long(80)
    ; first of two record groups
    RECSORT = long(0)
    ; there are two dimensions with uncertainties
    NUMREF = long(2)
    ; the uncertainties vary with longitude, therefore,
    ; AUXVAL.AVALS has one dimension
```

```
    NUMAPP = long(1)
    ; this covers the entire southern hemisphere
    START = long( [0, 0, 0, 0, 0] )
    ENDIT = long( [-1, -1, 45, -1, -1] )
    ; the uncertainties are floating-point numbers
    AUXFMT = long(67108864)
    ; the quantities are measured uncertainties
    AUXTYPE = long(262144)
    ; the units are in degrees
    UNITS = long(1745879298)
    ; there are no supplemental values
    NUMSUP = long(0)

    write,log_file_unit, RECTYPE, RECSORT, NUM-
REF, NUMAPP $
    , START, ENDIT, AUXFMT, AUXTYPE, UNITS, NUMSUP $
    , reserved, reserved

    ; set up the indices in the AUXRANGE record
    RECTYPE = long(81)
    ; first of two record groups
    RECSORT = long(0)
    ; the uncertainties refer to longi-
tude (the first index
    ; in the Level 1 dimensions) and latitude (the second
    ; index in the Level 1 dimensions)
    REFLEV = long( [1, 1] )
    REFNDEX = long( [0, 1] )
    ; the uncertainties vary over longi-
tude (the first index
    ; in the Level 1 dimensions)
    APPLEV = long(1)
    APPNDEX = long(0)

    write,log_file_unit, RECTYPE, REC-
SORT, REFLEV, REFNDEX $
    , APPLEV, APPNDEX, reserved, reserved

    ; set up the auxiliary information in the AUXVAL record
    RECTYPE = long(82)
    ; first of two record groups
    RECSORT = long(0)
    AVALS = U1

    write,log_file_unit, RECTYPE, RECSORT, AVALS
```

```
    ; set up the relevant elements for the second AUXSPEC
    ; record: longitude and latitude uncertainties in the
    ; northern hemisphere
    RECTYPE = long(80)
    ; second of two record groups
    RECSORT = long(10)
    ; there are two dimensions with uncertainties
    NUMREF = long(2)
    ; the uncertainties vary with longitude and latitude,
    ; therefore, AUXVAL.AVALS has two dimensions
    NUMAPP = long(2)
    ; this covers the entire northern hemisphere
    START = long( [0, 0, 46, 0, 0] )
    ENDIT = long( [-1, -1, -1, -1, -1] )
    ; the uncertainties are floating-point numbers
    AUXFMT = long(67108864)
    ; the quantities are measured uncertainties
    AUXTYPE = long(262144)
    ; the units are in degrees
    UNITS = long(1745879298)
    ; there are no supplemental values
    NUMSUP = long(0)

    write,log_file_unit, RECTYPE, RECSORT, NUM-
REF, NUMAPP $
    , START, ENDIT, AUXFMT, AUXTYPE, UNITS, NUMSUP $
    , reserved, reserved

    ; set up the indices in the AUXRANGE record
    RECTYPE = long(81)
    ; second of two record groups
    RECSORT = long(10)
    ; the uncertainties refer to longi-
tude (the first index
    ; in the Level 1 dimensions) and latitude (the second
    ; index in the Level 1 dimensions)
    REFLEV = long(1, 1)
    REFNDEX = long(0, 1)
    ; the uncertainties vary over latitude (the sec-
ond index
    ; in the Level 1 dimensions) and longitude (the first
    ; index in the Level 1 dimensions)
    APPLEV = long( [1, 1] )
    APPNDEX = long( [1, 0] )
```

```
    write,log_file_unit, RECTYPE, REC-
SORT, REFLEV, REFNDEX $
    , APPLEV, APPNDEX, reserved, reserved

    ; set up the auxiliary information in the AUXVAL record
    RECTYPE = long(82)
    ; second of two record groups
    RECSORT = long(10)
    AVALS = U2

    write,log_file_unit, RECTYPE, RECSORT, AVALS
```

Chapter 4

Pros and Cons

4.1 How Well Did We Meet Our Design Goals?

As described in Chapter 1, using a standard format has certain advantages and disadvantages.

The proposed format addresses the advantages in these ways:

Portability When written using XDR binary representations, a df dataset is completely portable across an electronic network. When using a native binary format instead, the TEST record at the beginning of the dataset tells just how the data were written, which is the first and most important step in converting it to be read on a different machine. (In fact, it is possible in principle to write generic programs to do the conversion automatically.) Furthermore, necessary metadata, such as dimensional information and identification of physical quantities and units, are written to the datasets as coded numbers, making the datasets portable across human languages as well.

Understandability All the information needed to accompany the data, such as descriptions of its dimensions, its "family history," special processing notes, and any sorts of general comments, have a specified place within each df dataset. Thus, the metadata are straightforward to find and interpret.

Reusability Because the df format is flexible and adaptable to a wide variety of needs, one avoids having to use dozens of different formats, all radically different. And while the absence of a software library of general df I/O subroutines hinders reusability of software at present, in the long term this difficulty should go away as such libraries are developed.

The df format also minimizes the disadvantages of standard formats:

Inflexibility The df format is quite flexible in being able to store many different kinds and forms of physical science data in whatever order or numeric representation the user desires, with processing descriptions and flags specifiable over any arbitrary subset of the data. While its most obvious application is in storing multi-dimensional rectangular gridded data fields, it can also store non-uniform grids, scattered point data, and other data objects. With the use of supplemental dimensional information, one should be able to store even non-rectangular collections of node-like data, such as trees or directed graphs (the DESCSUP record could hold the connectivity information).

Overhead The df format provides the user with the ability to choose how much overhead is involved in reading the dataset. That is, aside from a few records at the beginning of a data object, the amount of storage space required for a data set can be determined by the user, who can choose the numeric type used to store the data, as well as whether any packing or compression schemes are applied to the data. In addition, the user may choose which binary number representation is to be used, whether the portable XDR or the faster native binary formats.

Complexity In this item, the df format is clearly lacking. For a simple, straightforward data array, a df dataset can be fairly simple. The full format specification, however, is complicated and relies on concepts (such as mapping of dimension indices to data array indices) which many scientists will find hard to follow. The situation could be improved by the development of a standard library of subroutines to read and write the format.

Accessibility Instead of depending on a support group to develop all the software needed to use the df format, the user has the format specification itself and is thus free to implement it on any platform and any language desired. While this does not eliminate the need for a library of ready-to-use software, it does free the user from dependence on such software's existence.

Conformance It is possible to write a program which can scan a dataset and ensure that it does indeed conform to the standard.

4.2 Questions and Answers

Despite all its features, the df format does have some deficiencies. Some of these are justifiable (for the moment), some can be worked around (albeit awkwardly), and some must wait future enhancements of the format.

Why was this format created? It this really supposed to go into competition against the likes of HDF and netCDF?

No. In fact, the authors dream not of the day when the df format overwhelms all others, but of the day when the features they need—and which the df format at present alone supplies—show up in other, more widely accepted data formats.

Until that day comes, however, our research needs dictated a solution that could be used in the present, and so the authors had no choice but to design a new format. Taking a broader view, one can almost say that the df format was created as a means of demonstrating what we meant, and why the features we say we need are indeed important.

The df format specification is too complicated. As a scientist, I want something I can implement in BASIC in five minutes.

The format could have been made simpler, but only at the expense of making it more restrictive. The apparent complexity here is a result of being able to express complicated, pathological data structures; you can do things here you cannot do elsewhere. (If anything, df falls short of being able to represent every conceivable complicated data structure.) The moment we rule out such data structures, telling scientists they cannot write or think of their data in those terms, we lose.

In the short term, the complexity will frighten away some potential users. But in the long term, as a standard library of subroutines to handle the df format takes shape, the complexity will be hidden from the naive users, and they need never see this document. (The more advanced users, of course, will always be able to write their own software to improve performance in their individual environments.)

For now, the only consolation is to realize that for most cases, the specification is simpler than it appears. Simple cases can be simply expressed. The dimensional indices, for example, can be as straightforward or a twisted as the user cares to make them.

And, yes, a df format file can be written from a BASIC program!

It seems that a key feature of the df format is the use of numeric codes to represent metadata. This scheme, however, is not only overly restrictive on users creating their own data, but it relies on being able to distribute the definitions of these codes to all users of the format. And that seems unlikely to be successful.

Numeric codes are used instead of text strings for three reasons: (a) to save storage space (this means that we can require that metadata be present in each dataset, even in a massive collection of files), (b) to promote automated handling of the metadata (interpretation, labelling, cataloging, etc.), and (c) to reduce the ambiguity and language problems associated with the use of English text.

The codes, though, do necessitate implementing a mechanism for translating them (see Section B.2) as well as following the hierarchical scheme for adding new codes to the standard. Current technology is capable of supporting the distribution of the code definitions, whether it be through copying files or through the use of a distributed network server.

If a few small datasets are to be exported to another site, and there is some concern about whether that site is able to translate the codes, then we recommend that the code translations used in those datasets be inserted as COMMENT records into at least the first one. If large numbers of datasets are to be shipped, then it would probably be worthwhile to arrange for distribution of the complete set of code translations.

Note that the capability exists to define certain codes locally. One does not have to wait to write one's datasets until a central authority assigns a code; moreover, codes defined locally according to the standard are structured in such a way that *they will not conflict with other sites' local codes.* The only code that *must* be assigned centrally is the site identification code.

It should also be noted that, with the exception of the data format codes, translations are *not* necessary to read the data or determine its structure, but are merely an aid to interpreting it. If one knows, for instance, that a file contains temperature on latitude-longitude-pressure grids, then one need not rely on the codes to say so. Even the data format codes, in fact, are structured in such a way that a program should be able to parse an unknown code and figure out what data types are intended in most cases, without having access to a list of data format code translations.

The header records require the user to know what will be written before the data are written: the data sizes, the number of comments, the number and type of processing codes, etc. What if we want to write the data first, *then* figure out what the metadata are?

This is an unavoidable side effect of the requirement that programs written in languages such as Fortran know record lengths before the records are read. This requires that records be written in a certain order.

In the long run, though, as these languages move out of use, and this feature is less in demand, the requirement may be relaxed. In that case, records may be written in any order (so long as the TEST record comes first). At that time, it will be necessary to know the length of a data record which occurs before the metadata, in order to skip past it to read the metadata (which contains the information about how big the data records are); a new record type may then be created to give the length of the next record in bytes.

Using the df format, I can lump scalar quantities together as components of a single larger quantity. But I cannot do that with vectors or tensors (much less mix scalars, vectors, and tensors) without breaking them up into their components. I want to be able to nest Level 0 dimensions.

At present, single datum's complexity is limited. While the convention of the Level 0 dimensions provides for scalar, vectors, and tensor of arbitrary sizes and types, it does not allow for components of different forms (such as scalars and tensors) to be lumped together to form a single datum. Currently, separate data objects must be defined within a single dataset—a less than satisfactory solution.

What is needed is some completely general way of describing a single datum of arbitrary complexity. this is an area for future expansion.

Appendix A

Numeric Codes

A.1 Centrally Defined Codes

This section describes the various metadata codes that are centrally defined. These codes can be modified or added to only by the central administrative site, Site 1. The information given by these encodings is generally necessary to read the data, so these codes must be the same at all sites (with some provision for locally defined quantity codes and units codes.)

A.1.1 Data Format Codes

The data format codes are used in the OBJDESC.COMCOD, DESCRIP0.DATFMT, DESCRIP1.DESFMT, DESCRIP2.DESFMT, DESCRIP3.DESFMT, DESCRIP.DESFMT, PROCFORM.PRCFMT, PAKSPEC.DPAKFMT, PAKFORM.PAKFMT, and COMPFORM.COMPFMT record fields. These codes are integer values at least 32 bits long (because the codes are defined bit by bit, it is irrelevant whether these integers are considered to be signed or unsigned). The integer is expressed as a sequence of eight 4-bit nybbles, numbered from 0 to 7, the most significant nybble in the integer being number 0. The codes are structured in such a way that they can be interpreted by inspecting a hexadecimal printout.

The nybble specifications are as follows (LSB = Least Significant Bit, MSB = Most Significant Bit):

base	type						
0	1	2	3	4	5	6	7

0 : Numeric types

1 : logical or Boolean

0	1	size	true	false	0	0	0

size	0 : same size as short integer
	1 : same size as long integer
true-value	0 : LSB is clear (0)
	1 : LSB is set (1)
	2 : MSB is set (1)
	3 : MSB is clear (0)
false-value	0 : LSB is clear (0)
	1 : LSB is set (1)
	2 : MSB is set (1)
	3 : MSB is clear (0)

2 : byte

0	2	sign	0	0	0	0	0

sign	0 : unsigned
	1 : signed

3 : integer

0	3	sign	size	0	0	0	0

sign	0 : unsigned
	1 : signed
size	0 : short
	1 : long

4 : floating-point

0	4	size	0	0	0	0	0

size	0 : single precision
	1 : double precision
	2 : extended precision
	3 : super-extended precision

5 : complex floating-point

0	5	size	0	0	0	0	0

size	0 : single precision
	1 : double precision
	2 : extended precision
	3 : super-extended precision

1 : Character types

Note that the string lengths given below include only the characters in the string proper, not including any header or trailer bytes.

0 : XDR string

1	0	maximum string length

1 : fixed-length sequence of characters with no string length

1	1	length of array or string

2 : sequence of characters preceded by string length

1	2	maximum string length

3 : sequence of characters followed by a null character

1	3	maximum string length

4 : XDR counted string (IDL extension)

1	4	maximum string length

2 : Structure types

2	0	comp	code	0	0

comp — number of components in the structure

code — unique code to identify the structure

- 0 : a pair of bytes in (MSB, LSB) order which comprise a short integer
- 1 : a quadruplet of long integers (used as nodes for AUDIT.TREE)

The following are common examples of the data format codes:

Hex	Decimal	
01010000	16842752	logical with length of short integer (IDL logicals) true = 1, false = 0
02000000	33554432	unsigned byte
02100000	34603008	signed byte
03000000	50331648	unsigned short integer
03100000	51380224	signed short integer

03010000	50397184	unsigned long integer
03110000	51445760	signed long integer
04000000	67108864	single precision floating-point
04100000	68157440	double precision floating-point
04200000	69206016	extended precision floating-point
05000000	83886080	single precision complex
05100000	84934656	double precision complex
05200000	85983232	extended precision complex
05300000	87031808	super-extended precision complex
10000001	268435457	XDR string, a maximum of one character long
10000050	268435536	XDR string, a maximum of 80 characters long
11000001	285212673	fixed-length string of 1 character
11000050	285212752	fixed-length string of 80 characters
13000050	318767184	Null-terminates string; max length is 80 characters
20200000	538968064	short integer (signed) from a 2-byte base (MSB,LSB)
20300100	540016896	four unsigned long integers (used as nodes in the AUDIT.TREE structure)

A.1.2 Site Identifier Codes

It is anticipated that most sites using the df format will need to define and use their own local conventions for the integer codes used to indicate task IDs, special processing conditions, and data sources. Such local definitions can be managed and prevented from interfering with one another (when importing data sets from another site, for instance) by keying them to a unique site ID.

These site identifiers are long integer codes signifying where a dataset was created. To avoid duplication of codes, all site IDs are currently registered with the central administrative site, Site 1. Sites that are not registered with the main authority may use site ID 0, but their local conventions are virtually guaranteed to conflict with others'. Use of site ID 0, then, is strongly discouraged. (Note that site IDs are needed only by sites which generate datasets; one can certainly read datasets generated by others without having to acquire a site ID first.)

Site 1 is the site where this format originated: the Atmospheric Chemistry and Dynamics Branch (Mail Code 916) at the National Aeronautics and Space Administration's Goddard Space Flight Center, of the the United States Government. Site 1 will be the source of other site registrations and the source of updates and changes to the format. The current mailing address is:

Dr. Paul Newman
Code 916
NASA/GSFC
Greenbelt MD 20711
USA

email to: df@ertel.gsfc.nasa.gov

In addition to its use in separating local conventions, the site identifier code is also used in the audit tree of each dataset, specifying from where each data set referenced in the tree originates

Each code is an unsigned integer at least 28 bits long. The bits are numbered from 0 to 27, starting with the least significant bit. Bits 26 and 27 define an index of site types; the format of the rest of the integer code depends on the site type.

The bit format of the site ID codes is as follows:

0	typ	
28	26	0

typ (bits 26 and 27, values 0 to 3) type of site
0 : government
1 : university
2 : private industry
3 : other

0 : Government

0	0	country	agency	location	group
28	26	18	13	8	0

country	(bits 18 to 25, values 0 to 255) country of site
unit	(bits 13 to 17, values 0 to 31) government agency
location	(bits 8 to 12, values 0 to 31) location of site
group	(bits 0 to 7, values 0 to 255) group within site

1 : Universities

0	1	name	location	group
28	26	12	8	0

name	(bits 12 to 25, values 0 to 16383) university name
location	(bits 8 to 11, values 0 to 15) campus location
group	(bits 0 to 7, values 0 to 255) group on campus

2 : Private Industry

0	2	name	location	group
28	26	12	8	0

name	(bits 12 to 25, values 0 to 16383) company name
location	(bits 8 to 11, values 0 to 15) site location
group	(bits 0 to 7, values 0 to 255) group on-site

A few examples follow:

0:	Unknown
1:	U.S. NASA GSFC Chemistry and Dynamics Branch
65536:	U.S. NOAA NMC

A.1.3 Quantity Codes

The quantity codes are used in the OBJDESC.VARTYPE, DESCRIP0.VARTYPE, DESCRIP1.DESTYPE, DESCRIP2.DESTYPE, DESCRIP3.DESTYPE, and DESCRIP.DESTYPE record fields to describe what physical quantity the data or its dimensions represent. Each code is stored in an integer at least 32 bits long. This integer is expressed as a sequence of eight 4-bit nybbles, numbered from 0 to 7, the most significant nybble in the integer being number 0.

The quantity code format is as follows:

group							
0	1	2	3	4	5	6	7

group — is a broad collection of quantities that have some relationship to a particular discipline

00 :	General
01-0F :	Geophysical
10-1F :	Chemical
20-2F :	Astronomical
77 :	Site-defined quantities (explained in Section A.2.4)

General group

00	category	type	code

Atmospheric Science group

01	cat	type	code

Chemical Constituent group

10	family	type	code

Astronomical general group

20	category	type	code

A.1.4 Unit Codes

The unit codes are used in the DESCRIP0.UNITS, DESCRIP1.UNITS, DESCRIP2.UNITS, DESCRIP3.UNITS, and DESCRIP.UNITS record fields. It specifies the physical units for the data and its dimension. Each code is stored in an integer at least 32 bits long. The bits are numbered from 0 to 31, the least significant bit in the integer being number 0. Bits 29 to 31 define an index of eight possible forms of specifying units.

The format of the units code is as follows:

0 : Special and mathematical units

0	TYPE	CAT	UNIT
29	25	16	0

TYPE (bits 25 to 28, values 0 to 15) unit type
CAT (bits 16 to 24, values 0 to 511) unit category
UNIT (bits 0 to 15, values 0 to 65535) unit

1 : SI base units and dimensionless units from SI base units

1	BASE	D	EXPON	PREFIX	FAC	UNIT
29	25	24	20	12	4	0

BASE (bits 25 to 28, values 0 to 15) base unit (must be nonzero)

1 : length	$meter$	m
2 : mass	$gram$	g
3 : time	$second$	s
4 : temperature	$kelvin$	K
5 : amount of a substance	$mole$	mol
6 : electrical current	$ampere$	A
7 : luminous intensity	$candela$	cd
8 : plane angle	$radian$	rad
9 : solid angle	$steradian$	sr

D (bit 24, values 0 to 1) dimensionless unit flag
0 : specifies a single base quantity
1 : specifies a ratio of quantities with the same BASE (e.g., g/kg)

EXPON (bits 20 to 23, values 0 to 15) exponent of the base unit. Stored as 8+exponent (e.g., m^2 would have EXPON = 10)

PREFIX (bits 12 to 19, values 0 to 255) prefix for the base unit. Stored as 128+prefix (e.g., cm would have PREFIX = 126). Standard prefixes are:

-18	atto	a
-15	femto	f
-12	pico	p
-9	nano	n
-6	micro	μ
-3	milli	m
-2	centi	c
-1	deci	d
1	deka	da
2	hecto	h
3	kilo	k
6	mega	M
9	giga	G
12	tera	T
15	peta	P
18	exa	E

FAC (bits 4 to 11, values 0 to 255) for dimensionless quantities, the difference between the exponent of the numerator and the exponent of the denominator. Stored as 128+difference (e.g., km/cm would have FAC $= 128 + 5 = 133$)

UNIT (bits 0 to 3, values 0 to 15) for dimensionless quantities, provides an arbitrary index to distinguish between otherwise similar quantities (e.g., kg/g vs. g/mg)

2 : SI approved named units

2	BASE	PREFIX	CAT	UNIT
29	20	12	6	0

BASE (bits 20 to 28, values 0 to 511) base units. A bit array that specifies which fundamental SI quantities make up the unit. From LSB to MSB the bit position, the value to add, and the basic units are (bit number, integer value, unit):

0	1	length
1	2	mass
2	4	time
3	8	temperature
4	16	amount of a substance
5	32	electrical current
6	64	luminous intensity
7	128	plane angle
8	256	solid angle

(e.g., $N = kg\ m/s^2; kg = 2, m = 1, s = 4$ so that BASE $= 2 + 1 + 4 = 7$)

PREFIX	(bits 12 to 19, values 0 to 255) prefix for the base unit. Stored as 128+prefix (e.g., cm would have PREFIX = 126). Standard prefixes are given above
CAT	(bits 6 to 11, values 0 to 63) the category of the unit, grouping quantities with the same physical meaning (e.g., $newton$ and $dyne$ would be found in the same category: force)
UNIT	(bits 0 to 5, values 0 to 63) provides an arbitrary index to distinguish between quantities in the same category (e.g., $newton$ vs. $dyne$)

3 : General SI units in terms of base units with restrictions

4 : General non-SI units in terms of base units with restrictions

The structure of both of these forms is the same. Index 3 applies to the SI combination of units, and index 4 applies to any combination of units that include at least one non SI unit. There are certain restrictions in the sizes of the various fields, depending on the number of base units that make up the unit. Any unit that cannot be put into this form should use form 5 (General units without restrictions).

All cases have:

3 or 4	BASE
29	20

For 1 base unit:

E1	CAT	UNIT
15	8	0

For 2 base units:

E1	E2	CAT	UNIT
16	12	7	0

For 3 base units:

E1	E2	E3	CAT	UNIT
17	14	11	7	0

For 4 base units:

E1	E2	E3	E4	CAT	UNIT
17	14	11	8	5	0

For 5 base units:

E1	E2	E3	E4	E5	CAT	UNIT
18	16	14	12	10	6	0

For 6 base units:

E1	E2	E3	E4	E5	E6	CAT	UNIT
18	16	14	12	10	8	5	0

For 7 base units:

E	1	2	3	4	5	6	7	CAT	UNIT
	19		17		15		13	7	0

For 8 base units:

E	1	2	3	4	5	6	7	8	CAT	UNIT
		18		16		14		12	7	0

For 9 base units:

E	1	2	3	4	5	6	7	8	9	CAT	UNIT
	19		17		15		13		11	7	0

BASE (bits 20 to 28, values 0 to 511) base units. A bit array that specifies which fundamental SI quantities make up the unit. From LSB to MSB the bit position, the value to add, and the basic units are (bit number, integer value, unit):

0	1	length
1	2	mass
2	4	time
3	8	temperature
4	16	amount of a substance
5	32	electrical current
6	64	luminous intensity
7	128	plane angle
8	256	solid angle

(e.g., $N = kg\ m/s^2$; $kg = 2, m = 1, s = 4$ so that BASE $= 2 + 1 + 4 = 7$)

Ei the exponents of the base units. Note that a zero exponent is not included, since it is assumed that the base unit exists if specified. The size and value of the field is variable, depending on the number of base units as follows:

base num	bits	range	exponents
1	15 to 19	0 to 31	-16 to 16
2	12 to 19	0 to 15	-8 to 8
3	11 to 19	0 to 7	-4 to 4
4	8 to 19	0 to 7	-4 to 4
5	10 to 19	0 to 3	-2 to 2
6	8 to 19	0 to 3	-2 to 2
7	13 to 19	0 to 1	-1 or 1
8	12 to 19	0 to 1	-1 or 1
9	11 to 19	0 to 1	-1 or 1

CAT the category of the unit, grouping quantities with the same physical meaning (e.g., m/s and cm/s would be found in the same category: velocity). The size and value of the field is variable depending on the number of base units as follows:

base num	bits	range
1	8 to 14	0 to 127
2	7 to 11	0 to 31
3	7 to 10	0 to 15
4	5 to 7	0 to 7
5	6 to 9	0 to 15
6	5 to 7	0 to 7
7	7 to 12	0 to 63
8	7 to 11	0 to 31
9	7 to 10	0 to 15

UNIT — provides an arbitrary index to distinguish between quantities in the same category (e.g., m/s vs. cm/s). The size and value of the field is variable depending on the number of base units as follows:

base num	bits	range
1	0 to 7	0 to 255
2	0 to 6	0 to 127
3	0 to 6	0 to 127
4	0 to 4	0 to 31
5	0 to 5	0 to 63
6	0 to 4	0 to 31
7	0 to 6	0 to 127
8	0 to 6	0 to 127
9	0 to 6	0 to 127

5 : General units with no restrictions

2	BASE	T	CAT	UNIT
29	20	19	9	0

BASE — (bits 20 to 28, values 0 to 511) base units. A bit array that specifies which fundamental SI quantities make up the unit. From LSB to MSB the bit position, the value to add, and the basic units are (bit number, integer value, unit):

0	1	length
1	2	mass
2	4	time
3	8	temperature
4	16	amount of a substance
5	32	electrical current
6	64	luminous intensity
7	128	plane angle
8	256	solid angle

(e.g., $mph = 0.447\ m/s$; $m = 1, s = 4$ so that BASE $= 1 + 4 = 7$)

T — (bit 19, values 0 to 1) SI unit indicator

0 : SI unit

	1 : non-SI unit
CAT	(bits 9 to 18, values 0 to 1023) the category of the unit, grouping quantities with the same physical meaning (e.g., mph and fps would be found in the same category: velocity)
UNIT	(bits 0 to 8, values 0 to 511) provides an arbitrary index to distinguish between quantities in the same category (e.g., mph vs. fps)

7 : Locally defined units
These units are site-defined as specified in Section A.2.5.

A.1.5 Packing Codes

The packing codes are used in the PAKSPEC.CODE record field and specify the packing method for the data. Each code is stored in an integer at least 32 bits long.

Only two packing methods are currently defined. They both apply to gridded fields only. The first one has a code value of 1 and is called NMC scaling. This method packs data for each of the grid points of the last Level 1 dimension over all of the other Level 1 dimensions. For example, if the Level 1 dimensions consist of an array of $72 \times 91 \times 6$, then the data will be packed over the 72×91 values for each of the six grid points in the third dimension of the array. This method needs two parameters for each grid point to unpack the data. One parameter is the midpoint of the data, defined as

$$A = \frac{\max + \min}{2}.$$

The other parameter is a scaling factor, given as

$$n = \text{int}\,[\ln(\max - A)\ln(2)] + 1$$

The data are then packed by using

$$pdata = \text{round}\left[(data - A)2^{15-n}\right].$$

The packed data values and the parameter n are short integers. The parameter A is a floating-point number. These parameters are stored as (n, A) in the PAKVAL.INFO field. Note that n and A are vectors. To unpack the data, the reverse procedure is followed:

$$data = 2^{n-15} pdata + A.$$

The second packing method is somewhat similar to the first. There are three parameters. The first is a pointer into the Level 1 indices to indicate which dimension the packing applies. Using the example above, this value would be 2 (assuming a starting point of 0.) The next two parameters are very similar to the two parameters above. The first parameter is the minimum value of the data and the second is a scaling factor, given as

$$s = \frac{\max - \min}{32766}.$$

The data are then packed by using

$$pdata = \mathrm{round}\left(\frac{data - \min}{s}\right) + 1.$$

The packed data are short integers, while the two parameters are floating-point values. The parameters are stored as $(index, \min, s)$ in the PAKVAL.INFO field. Note that $\min$ and s are vectors. To unpack the data, the reverse procedure is followed:

$$data = s\,(pdata - 1) + \min.$$

The advantage to the first method is that more precision is retained than in the second method. The disadvantage is that exponentiation must be used in unpacking the data, which is a slow and computationally costly step.

A.1.6 Supplemental Codes

The supplemental codes are used in the DESCSUP.CODE record field. The code specifies the the interpretation of the supplemental information for the dimension descriptors. Note that this information is centrally defined rather than locally defined because it may be necessary in order to use the dimensions (e.g., the map projection parameters are needed if the dimensions are map coordinates).

There are currently four supplemental codes defined:

Code 1 specifies the map projection parameters: the radius factor (R), the map scaling factor (k_0), the pole longitude, and the pole latitude.

Code 2 specifies the map projection parameters: R, k_0, the pole longitude, and two standard latitudes.

Code 3 specifies the reference surface pressure for meteorological sigma coordinates.

Code 4 specifies the reference troposphere-stratosphere boundary pressure for meteorological sigma coordinates.

All supplemental values associated with the integer ID codes are floating-point numbers.

A.1.7 Compression Codes

The compression codes are used in the COMPSPEC.CODE record field and specify the method used to compress the data.

There are currently no compression codes defined, as compression has not yet been implemented. (Code will be defined and listed here as required.)

A.2 Locally Defined Codes

This section describes the various metadata codes which are locally defined. These codes can be created or modified by any site and generally refer to information which is useful in using or interpreting the data, but not strictly necessary in order to read it.

When giving datasets to other sites, it will be advisable to forward definitions of one's local codes as well. At the very least, these may be included as comments (in COMMENT records) in the first data file of a group, or, preferably, all the lists of local code definitions can be transmitted.

A.2.1 Task Codes

The task code is used in the AUDIT.TREE record field. It specifies the task that created the data. (Along with the site identifier and date, this information makes up the history trace for a file.) Task codes are integer values at least 32 bits long.

The codes may be assigned on any basis desired, although it is recommended that a particular system or format be devised to maintain order. One can assign a code for each general task, or for each program, or even for each set of run-time parameters for each program. How much information is required to distinguish one task from another is left to the dataset creator to decide.

A.2.2 Data Source Codes

The data source code is used in the OBJDESC.ISOURCE record field. It specifies the source of the data. This might refer to the site which generated a data set, but that site might have obtained the data from elsewhere. It might refer to the task which generated the data, but datasets created by two tasks which process the same data differently might still be considered to have the same data source. An instrument which took the data may be considered its source, or the platform from which the instrument operated. The idea of "data source," then, is fairly flexible and open to interpretation; it is therefore locally defined.

These codes are integer values at least 32 bits long.

An example may help in understanding the use of these codes. At Site 1 (NASA/GSFC Chemistry and Dynamics Branch), data are received from the National Meteorological Center (NMC) in their packed format. The data are then unpacked and put into the df format. The site identifier in the AUDIT.TREE record would have one node containing the site identifier for the NASA branch (1) and the task identifier for converting NMC packed data to the df format (16777216). The data source code would be associated with NMC (65536), since they provided the data.

In a second example, data are gathered by scientists at the same site for stratospheric aircraft missions. The site would again be the same (1), but the task of putting these data in the df format would be different from the first one (33554432). The data source would also be different and would indicate the NASA aircraft mission (512).

A.2.3 Processing Codes

The processing code is used in the PROCSPEC.CODE record field. It specifies how subsets of the data have been processed. This code is different from the task identifier, which identifies a task that was used to produce the dataset. Instead, the processing code provides additional information about the data itself; think of it as a sort of "Post-It" note attached to some subset of the data. For example, the lead time of forecasts, the interpolation parameters for subsets of the data, and parameters used in a model run can all be specified here. These codes are integer values at least 32 bits long.

A.2.4 Local Quantity Codes

As described in Section A.1.3, some quantity codes may be defined locally. These codes all begin with 77 as the first two hex digits. The specification of the last six hex digits is left to each site to specify. This allows sites to define special quantities that will not fit in any other categories. Also, these codes will not have to be registered through the central site, allowing for flexibility on the part of each site.

If a locally defined code proves to be widely used, a request should be made to Site 1 for its addition to the list of centrally defined quantity codes.

A.2.5 Local Unit Codes

As described in Section A.1.4, unit codes may be defined locally. These codes all begin with bits 29 to 31 as 111 (form 7). The specification of the other bits is left to each site to define. This allows for sites to define special units that will not fit in any other categories. The intention is that locally defined units codes will be used as a temporary measure while registration of an equivalent centrally-defined units code is pending. Also, these will not have to be registered through the central site, allowing for flexibility on the part of each site.

A.2.6 INFOSPEC Record Bytes

The INFOSPEC.INFO field contains a byte array that is specified by the data originator. Sometimes information must be included in a file for which none of the other records is suitable. When such information is human-readable, it should go into COMMENT records; when it is binary data or character flags, then it should go into an INFOSPEC record. The INFOSPEC.INFO field can contain anything; however, we strongly suggest that the first four bytes contain a long integer number that will uniquely identify what is contained after.

Appendix B

Implementation Notes

The df format specifies a standard form in which scientific data can be written, but it leaves to the user the decisions on how to implement programs to read and write the data.

This Appendix discusses issues relevant to implementing those programs; the procedures used at one site are described, and future enhancements are suggested.

B.1 Standard library routines

Compared with certain other proposed standard formats, one omission stands out with df: no attempt is made here to specify a standard library of I/O subroutines.

The fact is that a tradeoff exists between the flexibility needed to represent one's data as one desires, and the ability of a standard program to read such a dataset. Take, for example, the issue of bad-data flags: using a single bad-data flag value to signify bad or missing data over an entire dataset is relatively simple to include in a dataset reader subprogram; implementing a general reader able to cope with bad-data flag values which vary from region to region in the data (as specified by START and END indices) can be nightmarish. Let the reader be assured that such pathological cases are by no means exceptional or rare.

The temptation to develop a format standard in parallel with its software library is strong but should be resisted. It is all too easy, when confronted with the challenge of writing subroutines to handle the wide variety a good format allows, to rein in and limit that variety to make the programming task easier. Thus, for example, data arrays might be required to be written in row-major order, or their indices required to start numbering from 1.

In addition, the programming world is in a state of flux at present; object-oriented concepts are making inroads on traditional programming paradigms. Imposing a standard software library at this premature stage, allowing the format to be dictated by programming considerations, could end up being an example of imposed obsolescence.

The authors have decided, then, to concentrate on the format itself, making it as complete, flexible, and unambiguous as possible. The df format is exactly "A Standard

Format for Programmers to grovel in bits" (in contradiction to a slogan proudly proclaimed by the creators of another standard format). As experience with df datasets accumulates, a software library can evolve and be built up to meet users' needs.

In designing this format, flexibility was chosen over convenient uniformity. The task of programming for the general case is thus made more difficult, and a general library will take considerable skill to produce. This virtually rules out a single, standard, general set of subprograms able to read *every* df dataset appearing in the near future.

This strategy has the disadvantage of cutting off many potential users who are unwilling or unable to delve into the bit-level format specification to implement their own I/O software. In the long term, though, it prevents the format from being hobbled.

One promising possibility for the short term nevertheless appears when one notes that it is possible to discern from any df dataset in a mechanical fashion how it may be read. Thus, it should be possible to write a program which, after scanning a sample dataset, would then construct a subroutine to read any dataset similarly structured. Given that complicated arrangements of bad-data flag values, data packing schemes, and data array index orderings tend to be uniform across groups of datasets, one custom-generated subroutine could be used to read any of a large group of datasets, and a subroutine-writing program may prove quite profitable.

In the longer term, a general library of I/O subroutines for the df format is desirable and should be written. But the authors feel it is better to wait and let the best implementations rise to the top, than to impose an arbitrary software package with misfeatures everyone will later come to regret.

B.2 Numeric Codes

Metadata in the df format occupies comparatively little space and is largely independent of language because much of it is represented by integer codes. While the format standard defines no mechanism for translating between such codes and their meanings, how the translation is implemented can have a significant impact on performance.

On the one extreme, users could look up the codes themselves from long lists; this, of course, would be inconvenient and error-prone. At the other extreme, code translations might be obtained using a network server, similar to hostname/IP address equivalences found using the Domain Name System.

This section describes a simple implementation of the lookup mechanism based on a set of lists maintained in text files. This mechanism is used to provide translation for sites, tasks, quantities, units, packing codes, dimensional supplemental information (as specified in DESCSUP records), processing codes, and data source codes. Each code type has a corresponding file, and each code integer value is key value in a lookup table of plain text code explanations.

All of these files should reside in a central location (directory). If the system allows, there should be a global environment variable or logical name that should be set to *DFCODES*. This will allow software to be ported between systems, and the files can be found using the globally defined symbolic name.

Two categories of codes exist: those defined centrally, which are the same for everybody, and those which are defined locally by each site (See Appendix A). For example, site identification codes must be defined centrally, so that everyone will know which site is which.

The centrally defined code lists are given standard names:

SITEIDS.TXT	contains the list of site ID codes. Each line consists of two text fields delimited by a colon: *site-ID* and *full-site-name*.
VARTYPES.TXT	contains the list of physical quantity types. Each line consists of four text fields delimited by colons: *quantity-ID*, *four-letter-quantity-name*, *full-quantity-name*, and *preferred-units-abbreviation*.
UNITS.TXT	contains the list of physical units. Each line consists of four text fields delimited by colons: *units-code*, *full-units-name*, *units-abbreviation*, and *SI-units-definition*.
HOWPACK.TXT	contains the list of packing code explanations. Each line consists of four text fields delimited by colons: *packing-code*, *packing-description*, *packing-algorithm* (with PAKVAL packing parameters indicated as \$1, \$2, etc.), and *storage-spec* (showing how the packing parameters are ordered in a PAKVAL record, n indicating the number of parameter groups, S denoting a short integer, F denoting a floating-point number, and I denoting a long integer).
SUPCODES.TXT	contains the list of dimensional supplemental codes. Each line consists of three test fields: *supplementary-code*, *code-description*, and *parameter-list*. The *parameter-list* is, in turn, composed of subfields delimited by semicolons, each subfield describing a supplemental value found in DESCSUP.

One makes modification to these files at one's own risk, since modifications will put the file out of step with all other sites.

The other, locally defined code lists are in the files:

Unnnnnnn.VAR	contains a list of locally defined quantity codes; the format of this file is the same as for VARTYPES.TXT.
Unnnnnnn.UNT	contains a list of locally defined units codes; the format of this file is the same as for UNITS.TXT.
Unnnnnnn.TSK	contains the task ID codes. Each line consists of two text fields delimited by a colon: *task-code* and *task-description*.
Unnnnnnn.PRC	contains the processing codes. Each line consists of three text fields separated by colons: *processing-code*, *processing-title*, and *list-of-processing-variables*. The last field is composed in turn of an arbitrary number of subfields delimited by semicolons, each subfield describing a single processing variable expected in a PROCVAL record associated with a given processing code.
Unnnnnnn.SRC	contains the data source codes. Each line consists of two text fields delimited by a colon: *three-letter-source-abbreviation* and *full-text-name-of-source-institution*.

where "nnnnnnn" is the local site ID in hexadecimal.

Note that these file names consist of eight or fewer uppercase alphanumeric characters followed by a period and three more uppercase alphanumeric characters, making the file names portable to a very wide variety of computer systems.

A code translator program, then, would open one of these files, find the line with the code or text for which it is searching, and retrieve the text or code which corresponds to it. Since such translations need to be done only for human readability, calls to code translators will probably not be prevalent in production programs, and the inefficiency implied by this translation method is tolerable. One can improve performance, if desired, by hard-wiring some of the most commonly used codes at a site into look-up tables in the translator routines. If a code is not found in those tables, then the routine will go to the files.

Note that when a file (or, more likely, a set of files) is imported from another site, one would also import that site's Unnnnnnn. files as well. Because the "nnnnnnn" part of their names will be different, the local U files and the remote site's U files can co-exist without conflict. A df dataset reader can obtain the originating site's ID from the dataset and use it to find the local-code definition file it needs to make the translation into human-readable terms. For example, the task identifier file for Site 1 will be named U0000001.TSK, and the processing code file for Site 13579BD will be U13579BD.PRC.

B.3 File Naming Conventions

Using a standard file format in a conventional file-oriented computing environment, it is convenient to use a standard naming convention for the data files written in that format.

It should be noted that this naming convention is NOT a part of the proposed standard data file format, but instead co-exists alongside it as a separate entity which others may find useful.

Different systems, of course, have wildly different constraints on file names—lengths, extensions, legal characters, etc. The naming convention used should be adaptable for use with as many systems as possible.

The basic plan adopted here is to compose a file name of text fields separated if possible by delimiters. The first field should signify the physical quantity the file contains, and the last should indicate where the data came from. Each of the rest of the fields, included only if there is room and if the dataset creator desires, begins with a specific character distinguishing it from the others. (This allows for automated parsing of file names.)

B.3.1 Fields

The text fields which make up a file name are

quantity A four-character alphanumeric field which begins the file name and and indicates what physical quantity the file contains. This field must start with an alphabetic character.

date Indicates the date for which the data are valid. The leading character will be `I` (for instantaneous data), `D` (indicating data averaged over days), `M` (indicating data averaged over months), or `Y` (indicating data averaged over years). If the rest of the field is numeric, then it may be of the form 'yymmdd' or 'yymmddhh', where the y's stand for year digits, the m's for month number digits, d's for the digits of the day of the month, and h's for the hour of the day. If the part of the date field past the leading character is alphabetic, then it is a three-digit base-26 number (`A`=0, `Z`=25) indicating the number of days since 1 January 1970.

time Indicates the time for which the data are valid. The leading character is `T`. If the rest of the field is numeric, then it is of the form 'mmmmss', where the m's represent minutes, and the s's represent seconds.

source Indicates the source of the data. This field is a three-character alphanumeric string designating where the data came from. It goes at the end of the file name (usually as an extension) and has no leading character.

format Indicates the format or binary representation used in the file. The leading character is `X`. If there are no more characters in this field, or if only one character, `P` (indicating that the file contains packed data) follows, then the file uses XDR data formats. Otherwise, the lead character is followed either or both of two subfields: a `C` followed by either `A` or `E`, indicating whether character data is ASCII or EBCDIC; or a `B` followed by a floating-point format indicator (`E` for IEEE, `V` for VAX, `I` for IBM mainframe, and `Y` for Cray), a byte-order flag (`B` for big-endian, `L` for little-endian), a word-order flag (`B` for big-endian, `L` for little-endian), record-header flag (`C` for C-style stream files without record headers, `F` for Fortran record headers), a flag indicating the type of file record structure (`S` for pure stream files, `V` for variable-length records, and `F` for fixed-length records), and a packing flag (`P` for packed data, `U` for unpacked).

sequence Indicates some user-specified sequence number or code. The leading character is `E`.

gridtype Specifies on what sort of grid regular, rectangularly gridded data is written. The leading character is `G`, and the characters following should follow some local convention to indicate grid type, location, grid spacing, etc. If needed, decimal points may be represented by Level-2 delimiter characters (see Section B.3.2).

forecast Indicates, for model forecast data, how far in advance of the data-valid date or time the data were generated. The leading character is `F`.

special Is a field to hold any miscellaneous information the user feels should be included in the file name. The leading character is `S`.

B.3.2 Delimiters

Some character or set of characters is desirable to separate fields. In this file naming convention, a hierarchy of (non-alphanumeric) delimiter characters is defined, organized in decreasing order of magnitude at least three levels deep, with alternative characters defined for each level (to avoid operating system pickiness).

Many operating systems divide up file names into a name-part and an extension separated by a character such as a period. Therefore, we define the Level-0 delimiter as that character which separates between a file name proper and a file extension or type). The first choice for this character is whatever the operating system uses (a period for MS-DOS and VMS, a space for CMS, etc.). If an operating system does not use extensions, then a period should be used.

Level-1 delimiters are used for padding and as optional field separators. The first choice is the underscore character ("_"). If this is unavailable, the dash or hyphen character ("-") may be used.

Level-2 delimiters are used as decimal points. Periods may not be used in most situations, since they are the Level-0 delimiters in many environments. Instead, the first choice is the dollar sign ("$"), with the percent sign ("%") used as an alternate.

Note that Level-0 delimiters are required, while Level-1 delimiters are used to improve readability where the operating system allows longer file names. Level-2 delimiters may be used where needed.

B.3.3 Backus-Naur Form

This file naming convention may be specified in Backus-Naur form as:

```
<filename> ::= <name_part> <delim0> <type_part>

<name_part> ::= <quantity_field> [ <optional_field> ]
                { <delim1> <optional_field> }

<type_part> ::= <source_field>

<quantity_field> ::= <name_1> | <name_2> | <name_3> | <name_4>

<optional field> ::=  <date_field> | <grid_field>
                     | <forecast_field> | <sequence field>
                     | <format_field> | <special field>
                     | <time-field>

<date_field> ::= <ave_spec> <date_spec>

<grid_field> ::= "G" <grid_form>
```

```
<forecast_field> ::= "F" <forecast_form>

<sequence_field> ::= "E" <upperalphanumeric>
                     { <upperalphanumeric> }

<format_field> ::= "X"  { <format_form> }

<special_field> ::= "S" <upperalphanumeric>
                    { <upperalphanumeric> }

<time_field> ::= "T" <time_form>

<name_1> ::= <upperalphabetic> <delim1> <delim1> <delim1>

<name_2> ::= <upperalphabetic> <upperalphanumeric> <delim1>
             <delim1>

<name_3> ::= <upperalphabetic> <upperalphanumeric>
             <upperalphanumeric> <delim1>

<name_4> ::= <upperalphabetic> <upperalphanumeric>
             <upperalphanumeric> <upperalphanumeric>

<ave_spec> ::= "I" | "D" | "M" | "Y"

<date_spec> ::= <date_long> | <date_short>

<date_long> ::= <year_digit> <year_digit> <month_digit>
                <month_digit> <day_digit> <day digit>
                [ <hour_digit> <hour_digit> ]

<date_short> ::= <base_26_digit> <base_26_digit>
                 <base_26_digit>

<base_26_digit> ::= <upperalphabetic>

<grid_form> ::=  <grid_spec_char> { <grid_spec_char> }

<grid_spec_char> :== <upperalphanumeric> | <delim2>

<forecast_form> ::=   "H" <hour>
                    | "D" <day>
                    | <upperalphabetic> <digit> { <digit> }

<hour> ::= <digit> { <digit> }

<day> ::= <digit> { <digit> }
```

```
<format_form> ::=   "P"
                  | "C" <string_code>
                  | "B" <binary_code>

<string_code> ::= "A" | "E"

<binary_code> ::= <floating_format> <byte_order> <word_order>
                  <ftn_header> <disk_organize> <packing>

<floating_format> ::= "I" | "E" | "V" | "Y"

<byte_order> ::= "B" | "L"

<word_order> ::= "B" | "L"

<ftn_header> ::= "C" | "F"

<disk_organize> ::= "S" | "V" | "F"

<packing> ::= "P" | "U"

<upperalphanumeric> ::= <upperalphabetic> | <digit>

<upperalphabetic> ::= "A"-"Z"

<digit> ::= "0"-"9"
```

Appendix C

Future Enhancements

As of this writing, the df format is being used to store datasets in the Atmospheric Chemistry and Dynamics Branch at NASA Goddard Space Flight Center in Greenbelt, Maryland. A wide variety of datasets have been written in the format: vertical profile soundings, three-dimensional global grids of meteorological fields, global maps of total ozone, and data taken along aircraft flights. The format has been in use for about a year and has been used quite successfully.

In the near term, the important thing is to widen the scope of datasets written in the format, covering a larger variety of research groups and data structures. This should reveal any bugs or ambiguities in the format specification, and provide an opportunity to include any important missing features the authors have not thought of.

Future plans for the df format include:

- Make provision for using special file record structures. Because the records containing metadata are written to a dataset along with the data, dealing with random or keyed access records in a file is difficult. On systems in which files can be addressed by byte offsets within a file, random access is no problem, but for other systems it can be troublesome. Three solutions suggest themselves: (a) treating the metadata records the same as data records (e.g., by padding to a fixed record length); (b) creating a new POINTDAT record which points to a separate file in which the data are kept; and (c) reminding the user that this format does not specify how data are physically stored, but merely how the data are to be presented to the readers—the data and metadata, then, could conceivably be split into two distinct files which (through a layer of software) would appear to the reader subroutines as a single dataset. Whichever method is eventually chosen, care must be taken to maintain the highest degree of independence on any particular operating system.

- Be able to write records within an object in random order. Currently, the various records must be present in the dataset in a fixed and specified order. This will probably require creating a flag from one of the reserved words in the TEST record, as well as creating a new "Length-of-next-record" record to facilitate skipping around in a dataset.

- Determine a relationship to other formats. The two most widely used standard formats currently are the Hierarchical Data Format (HDF) [NCSA Software Tools Group, 1989], created by The National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign, and netCDF [Rew, 1990], created by the Unidata Program Center of the University Corporation for Atmospheric Research (UCAR). The HDF format relies on centrally-defined tags which indicate data objects within a dataset; a df dataset could be assigned its own tag and be encapsulated in an HDF dataset as an HDF data object. The netCDF format actually specifies a software interface rather than a dataset format. The model it uses to manipulate data is that of a rectilinear lattice of data points, which is a subset of the data structures dealt with by the df format. Thus, a software interface can in principle be written to deal with df format datasets using the netCDF library calls.

- Determine a representation for trees and arbitrary graph structures. The most likely method is to enter the data associated with nodes in the data records with a Level 1 or Level 2 dimension of "index," with the connectivity information being records in a DESCSUP record for that dimension. Alternatively, an Auxgroup may be defined to specify connectivity information between data points. A third possibility is to create a new record type specifically to represent connections between data points.

- Create standard I/O libraries.

- Create a suite of standard software tools for inspection of df datasets.

- Establish an appropriate procedure for assigning site IDs and registering other codes whose definitions are requested by various sites.

Appendix D

Miscellaneous Items

D.1 Trademarks

This document refers to many commercial products and companies. No endorsement of any of these is expressed or implied by their use here.

- Unix is a registered trademark of UNIX System Laboratories, Inc.
- VAX and VMS are registered trademarks of Digital Equipment Corporation.
- IDL is a trademark of Research Systems, Inc.
- Sun is a registered trademark of Sun Microsystems, Inc.
- Cray is a registered trademark of Cray Research, Inc.
- IBM is a registered trademark of International Business Machines, Inc.
- Iris and SGI are registered trademarks of Silicon Graphics, Inc.
- MSDOS is a registered trademark of Microsoft Corporation.
- Macintosh is a registered trademark of Apple Computer Inc.
- PostScript is a registered trademark of Adobe Systems, Inc.
- X Window system is a trademark of the Massachussetts Institute of Technology
- Motif is a trademark of the Open Software Foundation
- Post-It is a trademark of 3M Commercial Office Supply Division.

D.2 Acknowledgements

The authors would like to thank Dr. Mark Schoeberl of Code 916, NASA Goddard Space Flight Center in Greenbelt, Maryland, for supporting the idea of standard data formats and for allowing us to proceed to create a new standard when the existing ones were found not to meet our needs.

Bibliography

[NCSA Software Tools Group, 1989] NCSA Software Tools Group, *NCSA HDF Specifications*, University of Illinois at Urbana-Champaign, anonymous-ftp:ftp.ncsa.uiuc.edu, 1989.

[Pullen, 1990] Pullen, S. E., Recommended standard for seismic (/radar) data files in the personal computer environment, *Geophysics, 55*, 1260–1271, 1990.

[Ramirez, 1991] Ramirez, J. Raul, Understanding Universal Exchange Formats, *Photogrammetric Engineering & Remote Sensing, 57*, 89–92, 1991.

[Rew, 1990] Rew, Russel K., *netCDF User's Guide: An Interface for Data Access*, Unidata Program Center, anonymous-ftp:unidata.ucar.edu, 1990.

Index