SGP/38.3

Starlink Project
Starlink General Paper 38.3

Malcolm J Currie, P T Wallace & R F Warren-Smith

2009 December 10

Starlink Standard Data Structures

Contents

1 Introduction

This document should be read by programmers wishing to write applications software for Starlink. It describes various standard ways of arranging data, using the Hierarchical Data System (HDS). The most important of these is the NDF (Extensible *n*-Dimensional-Data Format), which is suitable for expressing a wide variety of data which occur in *n*-dimensional arrays—for example pictures, spectra and time series.

The early part of the document deals mainly with concepts, while later sections give more detailed definitions of the data structures.

Standard data structures comprise an important part of the *software environment*. Their use enables software packages to share data, thereby reducing the number of functions which must be duplicated and hence the total quantity of software needed.

To assist in designing standard data formats of adequate versatility, Starlink has implemented HDS, the *Hierarchical Data System*, a flexible system for storing and retrieving data in a structured fashion. HDS structures consist of assemblies of *data objects* arranged in trees. Each tree resides within a *container file*, one top-level tree per container file. Data objects have a *NAME*, which identifies that particular object, and a *TYPE*, which describes what sort of object it is. A data object can either be a *structure* or a *primitive*. Primitives are such things as integers, character strings, and floating-point numbers, and express the data themselves. Both structures and primitives can either be single items or multi-dimensional arrays. The overall philosophy is analogous to a computer's file system, where the files themselves (the equivalent of HDS primitives) are embedded within a hierarchy of directories (the equivalent of HDS structures).

Most people find the "tree" picture the most natural when visualising HDS objects. However, on some occasions it can be useful to regard an HDS object as a box (or *n*-dimensional array of boxes), marked with a NAME (unique at that level) and a TYPE (giving a clue to what might lie within). When the box is opened, further boxes are found. Eventually, you get to the items themselves (the primitives); all the rest is packaging.

Readers of the present document should be familiar with the HDS Programmer's Guide, SUN/92.

In conjunction with a release of the Starlink Software Environment (SSE—a forerunner of the ADAM system), Wright & Giddings (1983) proposed a number of standard data structures using HDS. The discussions which followed raised many new difficulties, and the proposals were never ratified. However, in the absence of anything more definitive, the proposed standard structures were used by several groups of implementors. During 1987 the discussion was re-opened following the selection of ADAM as the Starlink environment. The electronic correspondence on this topic runs into several hundred messages filling three bulky folders. It has not been easy to produce a design which will please everyone, for reasons that will be dealt with in the next section.

2 Overview

It has taken a long time and a large amount of effort to agree the design which is described here. The solutions to many apparently straightforward problems came only after protracted

discussion and thought. Many superficially attractive ideas proved on careful examination to be half-baked and unfortunately had to be dropped. Some of the arguments are subtle, and it must be acknowledged that not all contributors are fully convinced that the rejection of their ideas was justified. This is inevitable given the size and diversity of the Starlink community.

Design Fundamentals

The most fundamental issue (certainly the most fruitful source of controversy) is whether we are trying to design formats which are so comprehensive that any piece of information, however specific to an instrument or processing phase, has a defined location ready to receive it, or whether we are instead trying to design the simplest possible system which can do the job.

The first approach, sometimes called the "we've thought of everything" philosophy or *TOE*, is the one that has been traditionally employed. The designers of such systems have tried to predict everything that might be needed (in their experience as optical spectroscopists, aperture synthesists, X-ray observers, *etc.*) for the general case of a picture, spectrum, time series, or whatever. These designs generally work well for their inventor, but when others try to use them they find omissions, inconsistencies, ambiguities and limitations, and either have to add new items of their own or use existing items in a non-standard way. Even where a new form of data is expressed in what appears to be the standard way, experience has shown that precise interpretation by different application programs of all the various ancillary items (*e.g.* exposure time, astrometric parameters, *etc.*) cannot be relied on, and so these items become little more than comments.

The Starlink designs reject the TOE approach in favour of one where:

- The structures are simple.
- The processing rules for all components are defined.
- There is orderly treatment of extensions.

The third point—the treatment of extensions—is crucial. Most astronomer/programmers feel drawn to the familiar TOE approach, where there is a place to put the α, δ, exposure time, polarimeter setting, relative humidity, feed-horn collimation parameters, *etc.*, and are unhappy that many of the items they wish to include have to be "demoted" by being moved into an extension. Alternatively, they are willing to accept the need for extensions, but only for the idiosyncratic data required by other astronomers. It is important to understand that the extensions in the Starlink standard formats are an essential part of the scheme, safe havens where important but specialised items can reside, accessible to programs which understand them, and automatically copied from generation to generation. All extensions should be registered with the Starlink Head of Applications to avoid clashes between different groups of applications. Certain general-purpose extensions will be highly standardised, and will be used by many application packages.

The combination of (i) trying to keep the formats simple and (ii) defining precisely how the different items should be interpreted by application programs has produced a result which has remarkably little evidence of astronomy in it. This should not be regarded as a worry; the astronomical information, relating to astrometry, radiometry, timing, and so on, will reside in standard extensions which will be defined in due course. A byproduct of this conservatism

Table 1: Components of the Extensible *n*-Dimensional-Data structure

Component Name	TYPE	Brief Description
[VARIANT]	<**_CHAR**>	variant of the <**NDF**>
[TITLE]	<**_CHAR**>	title of [**DATA_ARRAY**]
[DATA_ARRAY]	<**various**>	NAXIS-dimensional data array
[LABEL]	<**_CHAR**>	label describing the data array
[UNITS]	<**_CHAR**>	units of the data array
[VARIANCE]	<**s_array**>	variance of the data array
[BAD_PIXEL]	<**_LOGICAL**>	bad-pixel flag
[QUALITY]	<**various**>	quality of the data array
[**AXIS**(NAXIS)]	<**AXIS**>	axis values, labels, units and errors
[HISTORY]	<**HISTORY**>	history structure
[MORE]	<**EXT**>	extension structure

(which came largely from the need to reduce the task to a manageable size) is that the standard structures, and the general-purpose applications which process them, may have uses outside astronomy.

The Extensible n-Dimensional-Data Format

The Starlink standard data structures can be divided into two categories: *low-level,* and *composite*. Low-level structures are self-contained; coupled with individual data objects they are used to build the composite formats, and include axis information, title, history, and quality. The composite formats are akin to the Interim Environment's Bulk Data Frame (see SUN/4).

Since the idea of some completely general data format has been rejected (for reasons already presented), even for astronomical data, a number of composite data formats will be defined for various classes or forms of data as required.

The only current example of a composite data format, the NDF, is presented in Table 1. (NDF is short for *Extensible n-Dimensional-Data Format* and will be described fully in Section 11.) The NDF is based on the *n*-dimensional data array, which is the most natural way to express many sorts of astronomical data set—notably spectra, pictures and time series.

Within an HDS container file, the NDF structure resides usually, though not necessarily, at the top level. For example, the top-level structure within a container file might be built from several NDFs, each an observation of the same source but through a different filter.

(The < and > signs are not actually part of the TYPE, nor are the brackets around the NAME. They are just a notation convention to distinguish TYPE from NAME.)

There are several low-level objects within the NDF, each with a *NAME* for access and identification, and a *TYPE* to control the general processing. They comprise:

- primitives: [**TITLE**], [**LABEL**], [**UNITS**], [**BAD_PIXEL**],
- structures: [**HISTORY**], [**MORE**];
- arrays of structures: [**AXIS**];
- objects which can be either primitive or a structure: [**DATA_ARRAY**], [**VARIANCE**], [**QUALITY**].

Only the [**DATA_ARRAY**] is mandatory—so a primitive HDS object containing a 2-D array of numbers (for example) is valid as the only component of an NDF. A full description of the components is given in Section 11 and the meaning of the TYPEs in Section 3.

The omission from the NDF of the maximum and minimum values of the data array, and other quantities which can be deduced from the data, is deliberate. This is because experience has shown that sooner or later applications come along which fail to keep these numbers up to date.

In designing the NDF, efforts were made to retain some compatibility with the original Wright-Giddings proposals. A limited but useful degree of compatibility was achieved—the former location of the main data array (*i.e.* at the top level) is still recognised, and the name has been retained—but it proved impossible to accommodate more of the original proposals without enormously adding to the complexity of the processing rules.

As was mentioned earlier, the NDF is a simple structure devoid of any obviously astronomical items but which can be used to express many different varieties of astronomical data. IPCS spectra, CCD pictures and Taurus datacubes, for example, are all essentially *n*-dimensional arrays together with some ancillary items, and fit naturally into the form of the NDF, simple though it is. In the next section, we will look at the general question of **layering**—how the elaborate requirements of any particular data type can be broken down into different levels of generality. We will then go on to consider the topics of **substructures** (how common building blocks may be identified and exploited), **extensibility** (how items peculiar to a data type or an application package may be accommodated), and various processing considerations including the **propagation** of extensions.

Layering

There are several well-defined levels of generality in designing data formats, with each level building on those below it (see Table 2). Having designed and implemented HDS, the Starlink Project could have left it at that and, beyond stipulating that applications must use HDS for their data storage, have allowed each software package to be autonomous. This is the **HDS** level. The second level uses a **mathematical** representation to provide generality; the rules for

Table 2: Layering of Starlink data formats.

specialist
astronomical
mathematical
HDS

processing data objects are mathematical abstractions, though chosen to map well onto the processing of pictures and spectra and other types of astronomical work. The **astronomical** representation is much more complex than the mathematical level, and contains information relating to astrometry, radiometry, *etc.* **Specialist** structures for instrument- or application-specific processing are still more complex.

The HDS-level approach was rejected because it fails to ensure the required degree of compatibility between application packages. The astronomy level, on the other hand, could not have been defined properly in the time available, and the highest level will, of course, only be supplied by the authors of specialist packages. Therefore, Starlink initially selected the mathematical representation. As well as offering the possibility of a useful standard in a reasonable time (it will be much more capable than INTERIM's BDF, which has nonetheless proved extremely successful and versatile), concentrating on the mathematical level makes it easier to identify a subset of common low-level data objects. Because the processing rules are well defined, software to handle these low-level objects is relatively easy to write, and once written can be used by many different packages.

KAPPA (**K**ernel **App**lication **Pa**ckage — see SUN/95) is a set of applications which processes these abstracted data objects. KAPPA is intended to be a quick, exploratory data-analysis tool, and its applications will act as paradigms or templates for specialist software packages.

Another important package is FIGARO (SUN/86), which has been influential in defining the Starlink standard data formats and is at present undergoing a refit to bring it fully into line. FIGARO is a large and mature set of picture processing and spectral applications, and though not as formally correct as KAPPA will be the dominant general purpose ADAM application package for some time to come and will have a profound influence on the design of other packages.

Standard software interfaces will be written to access the low-level structures. They are currently being designed and will be described in a future version of this document.

Substructures

The essence of HDS is the ability to define multiple levels rather than having everything at the top level in a "flat" design. Substructures make it easy to copy or delete parts of a structure, and provide flexibility and extensibility. Some implementors mistrust multiple levels of hierarchy, and have advocated the use of flat arrangements, combined with elaborate naming schemes (with wildcarding) to distinguish between different classes of object. However, this approach has been discredited as inefficient and arcane, and has not been used in the Starlink standard formats. It is also deprecated within NDF extensions.

Table 3: Components of <**ARRAY**> Structure, [**VARIANT**] = 'SIMPLE'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SIMPLE'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DATA**]	<**narray**>	actual value at every pixel

Table 4: Components of <**ARRAY**> Structure, [**VARIANT**] = 'SCALED'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SCALED'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DATA**]	<**narray**>	scaled numeric value at every pixel
[**SCALE**]	<**numeric**>	scale factor
[**ZERO**]	<**numeric**>	zero point

Low-level structures should be kept as small and simple as possible. They should contain related objects, whose meanings are defined and whose processing rules are specified. If a structure is created which contains unrelated objects all lumped together, it will be unwieldy, the processing rules will be restrictive or highly complex, and it will be a difficult object for programmers to manipulate. Furthermore, modular substructures can be used in different data formats without duplication of software. The conventions for the interpretation of a structure, as well as its components, must be defined if there is to be compatability between different software packages.

On the other hand, over-elaborate structuring (structure which would demand more work of the applications programmer and reduce the efficiency of applications) should be avoided. For example, it would probably be inefficient to represent a celestial position with separate components for hours, minutes, seconds and degrees, arcminutes, arcseconds, rather than expressing the two angles as floating-point numbers or using a single character string (notwithstanding the example in SSN/27, which is a demonstration rather than a recommendation!).

Example structures are shown in Tables 3–5. The first two are *variants* of the <**ARRAY**> structure, and are different ways of storing an *n*-dimensional array of numbers. A variant qualifies the TYPE and the processing rules of the structure, and may appear in any structure. The most basic form is specified by [**VARIANT**] = 'SIMPLE', which is the default should [**VARIANT**] not be present. In this case <**ARRAY**> comprises an array of numbers plus the origin in each dimension. [**ORIGIN**] defines the zero point of the pixel coordinate system. For [**VARIANT**] = 'SCALED', the array of numbers has been scaled, perhaps as 16-bit integers to save disk space for large-format data. [**SCALE**] and [**ZERO**] are used to generate the actual array values. In almost all cases, standard subroutines will deal automatically with the different variants and simply present the application with a locator to an array of numbers.

All the objects in the first two examples have primitive TYPEs. This need not be the case. The <**HISTORY**> structure contains a further structure [**RECORDS**] to store history text and time tag. History records are brief and are only intended to assist the user. Their contents must **not**

Table 5: Contents of the <**HISTORY**> Structure

Component Name	TYPE	Brief Description
[**CREATED**]	<**_CHAR**>	creation date and time
[**EXTEND_SIZE**]	<**_INTEGER**>	increment number of records
[**CURRENT_RECORD**]	<**_INTEGER**>	record number being used
[**RECORDS**(*m*)]	<**HIST_REC**>	array of *m* history records

be used by applications to control processing.

The rules for designing new structures are presented in Section 13.

Extensibility

No data format design will ever satisfy all requirements, and some provision has to be made for accommodating ancillary information, specific to a group of applications or a particular instrument for example. What should an application do if it encounters objects which are not part of a standard structure?

In most of the ASPIC applications (see SG/1) such objects — which have to be expressed in simple ways akin to FITS descriptors — are simply ignored, and consequently do not appear in any new data frames created by the applications.

A hierarchical data system like HDS enables all these specialist data to be expressed in natural ways and to be attached to the main data structure at appropriate places within the structure. During processing these extension substructures can be copied to output data structures. Of course, as a result of the computations there may then be inconsistencies between the specialist and the general data objects; this is unavoidable. Rules will have to be laid down about what applications can be run on what types of data and in what order, and sometimes it may be necessary to resort to utilities which edit HDS structures to fix up inconsistencies. Specialist packages could implement all this transparently by providing command procedures.

In the NDF, extensibility is provided through the [**MORE**] structure. Information required by application packages (and under the complete control of those packages) is arranged into structures usually of TYPE <**EXT**> and stored within [**MORE**]. In order to allow applications to recognise their own extension objects without risk of clashes, the names and types of the extension structures must be registered with the Head of Applications. An example of an extension might be one called ASTROMETRY, which would contain information about the relationship between the data and the celestial reference frame.

[**MORE**] structures can occur once at the top level of the NDF, and once in each [**AXIS**] element.

As well as processing the extensions they recognise, applications are obliged to propagate all other extensions to any output structures.

Details of the defined extensions are given in other documents.

Propagation

Although the data objects in the NDF are general, not all applications will know how to process all the objects. For example, the [**VARIANCE**] in the NDF becomes meaningless after

thresholding. Therefore, the rule for propagation, is as follows.

- Applications always propagate extensions.
- An application may only process and propagate a data object whose processing rules it understands and can fully implement, in order to avoid rendering the object meaningless or wrong. Unknown or poorly understood objects must be ignored—never fudged.

Tree walking

"Tree walking", *i.e.* by one means or another moving to a higher position within an structure and processing data objects there, is forbidden. If such objects are required their names must be acquired from outside the application, and a new locator obtained.

Instrumental data

It will often be most convenient and efficient to delay conversion of instrumental data into standard formats until calibration, reformatting and other preprocessing operations have been completed. The software for performing this conversion is best provided by the instrument builders. However, in some instances it will prove convenient to use a composite structure (*e.g.* an NDF) for uncalibrated data so that general-purpose applications can be used for for display and other quick-look operations.

Bad data

Two methods are available for dealing with bad data. The **magic value** method uses special values stored within the data themselves to indicate that a datum is undefined or invalid. The second method is **quality**. Each data value may have associated with it a data-quality value, an 8-bit code which flags whether the data value is special in some way or combination of ways.

The NDF has a [**BAD_PIXEL**] flag, which allows applications to find out in advance if there are any magic-value data within the [**DATA_ARRAY**]. If not, a version of the algorithm which does not do the checks can be invoked, in order to save time. Note that in this document the term *pixel* is used in its generic sense, *i.e.* equivalent to an *array element* or *bin*, and therefore, *pixel* does not imply membership of a two-dimensional array.

Full details of bad-pixel methods are presented in Section 7.

3 Naming, Types and Variants

3.1 Character Set

NAMEs, TYPEs and the contents of strings in HDS **must** consist only of printable ASCII characters (*i.e.* hexadecimal 20 through 7E). Beyond this, HDS itself (see SUN/92) imposes restrictions on NAMEs and TYPES, to which we add the requirement that the first character of a (non-primitive) TYPE must **not** be underscore.

It is **strongly recommended** by Starlink that NAMEs and TYPEs be limited to letters, numbers, and the underscore character, and that the first character be a letter. This is to prevent inconvenience to users, who will find themselves having to resort to special syntax (extra quote marks, for example) in order to resolve command-line ambiguity where unconventional NAMEs or TYPEs are present. Worse, the possibility cannot be ruled out that unexpected interactions between command-language features and imaginative NAMEs or TYPEs will, at some stage, cause insurmountable difficulties.

The names of HDS container files should be chosen to be the same as the names of their top-level data structures, or at least to be closely related. Note that filenames must **not** appear in applications code (see SGP/16).

3.2 The Rôle of TYPE

TYPE describes the form of a data structure and the general rules for interpreting or processing all structures of that form. The NAME identifies the particular object. The TYPE tells the program what to expect and how to interpret the object.

It is possible to take the view that the TYPE attribute of HDS objects is important only for primitives (*e.g.* <**_REAL**>) and is not needed for higher-level structures. Such structures would thus have a NAME but no TYPE, and applications would interpret them by the presence or absence of NAMEd components, as well as by data values within the structure. Note, however, that the concept of type will be needed even if the TYPE facility is not itself used. TYPE is already available for this purpose and its use eliminates the need to define naming schemes or other conventions, which could differ between packages and ultimately cause clashes. It is supported directly by the HDS subroutines and is likely to be more efficient than other methods. It assists both the user (who will see it in a standard place when structures are listed) and the programmer (who can expect a structure of registered TYPE to conform to a given set of rules, and to have associated interface routines). A further objection to naming schemes comes from the limited length of HDS names (15 characters, chosen to maintain adequate storage efficiency).

In order to minimise the total number of TYPEs to be recognised, the TYPE may be supplemented by a *variant*, an HDS object of TYPE <**_CHAR**> called [**VARIANT**]. In such cases, the TYPE will define the general processing rules for a given structure, while [**VARIANT**] will specify detailed interpretation. The variant also enables a data structure to be developed and modified. Therefore, applications must ascertain the value of [**VARIANT**] in every structure to determine whether or not it can process that version of the structure. Mostly, this will merely be a check of the existence of [**VARIANT**].

To avoid incompatibilities between programs, it will be necessary to register both TYPEs and variants with the Starlink Head of Applications.

How do you decide whether a structure should have a new TYPE or be a variant of a TYPE? The guideline is as follows. [**VARIANT**] should be used if an algorithm written to process an existing TYPE could also process the new structure, merely by interposing an automatic conversion routine which requires no additional external information. If [**VARIANT**] is not present within a structure, or it has the value `'SIMPLE'`, then the structure is in its most basic and readily comprehensible form.

Any general utility program, (*e.g.* one from the KAPPA package) can use TYPE when available, but must also be prepared to rummage within a data structure and identify the components it needs by other methods, if the particular TYPE is not recognised.

3.3 Notation and Pseudo-types

A component of an HDS structure is characterised by its dimension(s), NAME, TYPE and meaning. The notation adopted in this document to describe these attributes is as follows.

The **dimensions** are given in a FORTRAN-like way. For example:

[**DATA_ARRAY**(NAXIS1,NAXIS2)]

is an NAXIS1 × NAXIS2 array with name [**DATA_ARRAY**]. The square brackets are a notation convention in this document to indicate the NAME of a data object. They are not part of the NAME. As in this example, use is made of symbolic constants (*e.g.* NAXIS1) in specifying array dimensions. The meaning of these constants are described for each structure.

A **NAME** in capitals is the NAME the component must take; where the NAME is in lowercase it is generic, *i.e.* it may be chosen by the programmer provided it does not duplicate a Starlink-reserved NAME. NAME is not globally significant—its meaning is only defined within the context of a containing structure.

TYPE may be one of the *primitives* or one allotted to a *structure*, denoted <**_TYPE**> and <**TYPE**> respectively. As mentioned earlier, the > and < signs form a notation convention used in the text of this document to signify the TYPE of a data object, and they are not themselves part of the TYPE. Details of HDS primitive TYPEs can be found in SUN/92.

Within Starlink standard data structures TYPE is recognised globally—a given TYPE always means the same thing no matter where it appears in a structure.

In order to allow for flexibility in the way in which data are represented, certain structure components are allowed to take one of a number of options. This is indicated in this document by "pseudo-types", which are denoted by <**type**>; the standard pseudo-types are listed in Table 6.

The <**various**> notation is used where the options do not neatly fit into one of the categories in Table 6. Sections 10 and 11 contain tables of the contents of standard structures, followed by descriptions of the individual items. The possible options for each <**various**> can be found within the description of the associated data object.

There is problem for the notation. Certain data objects are arrays. The number and sizes of each dimension of these arrays are mostly not fixed, and so cannot be specified explicitly in a structure's table of contents. To overcome this problem, the array pseudo-types <**narray**>, <**farray**> and <**iarray**> are used instead of the scalars <**numeric**>, <**float**> and <**integer**> respectively in cases where the number and size of an array's dimensions cannot be specified *a priori*. Thus, if a vector object has its dimensions specified in a table, it will not have an array pseudo-type assigned to it, because the dimensions are constant.

The first letter of the last-three types in Table 6 may be an *aide mémoire* of their meanings: **c** stands for complex, **p** for primitive and **s** for scalar.

The notation <**c_array**> can mean either an <**ARRAY**> or a <**COMPLEX_ARRAY**> (described below). The <**ARRAY**> TYPE illustrates some of the terminology used in this document. <**ARRAY**> has POLYNOMIAL, SCALED, SPARSE and SPACED variants. These forms are all methods for expressing an array of numbers, intended for use where a primitive array would not be suitable. Whenever one of these special forms is used, the primitive TYPE of the equivalent

Table 6: Standard Pseudo-types

Pseudo-type	Allowed Types
<**various**>	structured or primitive (to be specified)
<**numeric**>	one of the primitive numeric TYPEs
<**integer**>	one of the primitive integer TYPEs
<**float**>	one of the primitive floating-point TYPEs
<**narray**>	a <**numeric**> array
<**farray**>	a <**float**> array
<**iarray**>	a <**integer**> array
<**c_array**>	an array of complex numbers or an <**ARRAY**> TYPE
<**p_array**>	a <**narray**> or an <**ARRAY**> TYPE
<**s_array**>	a <**numeric**> scalar or <**p_array**>

array (for example _REAL), which we will call the *equivalent primitive type*, must also be specified to allow application programs to select the most efficient form of processing.

Some examples to clarify the meanings of the pseudo-types are presented in Table 7. Parameterised dimensions indicate that their number and sizes are fixed. Numerical dimensions are arbitrary.

4 Labels and Units

Often, data values stored in structures will be accompanied by textual labels describing what they are and what units they are represented in. Whilst this paper describes abstracted data structures, clearly the main use of the data formats is for astronomical data. Therefore, it is important that the form and content of these label and units strings conform to a definite standard, so that they are readable and unambiguous.

One important reason for consistency is so that those general-purpose applications which have more than one input data array can test for equality the units of the various associated data objects. For utility operations, like addition and subtraction, the application must warn the user if the units are different, and where an output object is being generated must drop the units altogether. In other cases it may not be possible to proceed with the processing at all.

A minor reason for a rather definite standard is that future applications (but not the present ones), may have the capability of interpreting and processing labels and units. For example, consider a hypothetical application which would calibrate an array of data by dividing into it another array containing the calibration information. The attributes of the two arrays, and the result, [**FLUX**] might be:

Table 7: Pseudo-type Examples

Entry in a Table of Contents		Example	
Name	Pseudo-type	Name and dimensions	Type
[BASE]	<**numeric**>	**[BASE]**	<**_INTEGER**>
[LIST(NAXIS,NDATA)]	<**integer**>	**[LIST**(2,450)]	<**_WORD**>
[TMIN(NAXIS)]	<**float**>	**[TMIN**(3)]	<**_DOUBLE**>
[DATA]	<**narray**>	**[DATA**(512,100,3)]	<**_REAL**>
[VARIANCE]	<**farray**>	**[VARIANCE**(100)]	<**_DOUBLE**>
[DATA_ARRAY]	<**iarray**>	**[DATA_ARRAY**(512)]	<**_INTEGER**>
[DATA_ARRAY]	<**c_array**>	**[DATA_ARRAY]** *or* **[DATA_ARRAY]**	<**ARRAY**> <**COMPLEX_ARRAY**>
[QUALITY]	<**p_array**>	**[QUALITY**(384,512)] *or* **[QUALITY]**	<**_UBYTE**> <**ARRAY**>
[WIDTH]	<**s_array**>	**[VARIANCE]** *or* **[WIDTH**(2000)] *or* **[WIDTH]**	<**_REAL**> <**_INTEGER**> <**ARRAY**>

Name	Label	Units
[DATA]	`flux`	`count/s`
[CALIB]	`flux-calibration`	`J/(m**2*Ang*count)`
[FLUX]	`flux-density`	`J/(m**2*Ang*s)`

where `Ang` is Ångstrom to avoid a clash with `A` (ampere), and parentheses are to bracket units in the denominator; `J/m**2/Ang/s`, for example, would be easy to misinterpret. In this case (and probably all others), it would be impossible to predict the label to be associated with the result. However, the units could (with some care) be determined. Towards this goal, present-day applications should conform to the following guidelines when generating the output **[UNITS]** object as follows.

- Fortran conventions and intrinsic functions are to be used.
- Parentheses should be placed around the input **[UNITS]** if an intrinsic function or arithmetic operation has been applied to the associated data.
- If a meaningless output **[UNITS]** arises, say due to addition or subtraction of different input **[UNITS]**, no output **[UNITS]** should be created.

To clarify these rules here are some examples.

1st input **[UNITS]**	2nd input **[UNITS]**	Operation	Output **[UNITS]**
`'count/s'`		arithmetic with a constant	`'count/s'`
`'count/s'`		logarithm to base 10	`'log10(count/s)'`
`'count/s'`	`'s'`	multiplication of data	`'(count/s)*(s)'`
`'count/s'`	`'W'`	subtraction of data	—
`'J/(m**2*Ang*s)'`		exponentiation to power of 2	`'2.**(J/(m**2*Ang*s))'`

Present applications will not have the ability to interpret or parse **[UNITS]** objects. Until standard routines appear which do this (which will not happen soon and may never happen), applications must not attempt to do this themselves.

The standards for saying which units are to be used for each kind of value will probably have to be determined by a group of interested users and implementors. However, a start can be made by looking at the scheme proposed along with the FITS specification (Wells & Greisen, 1979). The FITS scheme may be summarised as follows:

- Consistent with the International System of Units (SI).
- Add "degrees" for angles.

- Add Janskys (Jy) for flux density.

The first proposal—conformity with SI — includes the standard prefixes used to scale values by factor multiples of 1000. There is one problem here: micro- is abbreviated to μ, which is not a character available within HDS strings (see Section 3.1; the character `u` should be used instead. Also, the SI units include an Ω abbreviation; here, the full unit name, `ohm`, should be used. Using SI units is, on the face of it, clean and simple, and the right thing to do. However, at present, SI units are quite alien to many in the astronomical community, and their adoption as a rigid Starlink standard would probably not be acceptable. (There is also the complication that to distinguish the abbreviations for seconds and siemens case-sensitive testing would be required.)

Some of the SI quantities are rarely, if ever, used in astronomy, but are included in Table 8 for completeness.

Within the literature there is liberal misuse of the terms *intensity*, *flux* and *flux density*. For example, "flux" is frequently used where "flux density" is the correct term. Table 9 shows the diversity of units currently employed, and the duplication of abbreviations, *e.g.* `m` both for minute and metre. Starlink's current recommendation is to use SI units; if this is unacceptable, the names or abbreviations used must be unambiguous.

5 Errors

"Error" is a woolly concept; it means different things to different people, and is generally intimately tied to specialist data or applications. Even where some mathematical description is adopted, with rigid rules describing the effects of different operations on the error values, there is still no way of protecting the user against invalid processing sequences and, consequently, the generation of incorrect and misleading error estimates. However, there have been repeated and emphatic demands for there to be some provision for error handling, so that (at the very least) error bars can appear on plots.

The compromise adopted in Starlink data structures is to allow normal statistics to be assumed and to provide for variances to be stored along with the data. User-defined structures may employ different representations of error information.

On input to an application assume that the elements of the [**DATA_ARRAY**] data object are independent and subject to normal statistics, and that the contents of the [**VARIANCE**] data object are the variances of the corresponding elements of [**DATA_ARRAY**]. For most data, however, this will not be true, and therefore the variance should be taken merely as a guide. Ultimately, it will be entirely the responsibility of the user to ensure that the result is sensible. For example, if two copies of the same data are added together this will not be detected by the application, and the variances will be wrong (by a factor of two).

The [**VARIANCE**] data object will be propagated in cases where the application can readily compute its processed values. For example, it is relatively easy to define the effect on the variances of simple scalar and vector arithmetic operations, and so variances will be computed and included in output structures; however more complicated operations, including convolution, are not so amenable, and variances will not be computed. The programmer should state for each program whether [**VARIANCE**] is processed or not, and the limitations of the variance computation.

Table 8: International System (SI) Nomenclature

Physical Quantity	Name of Unit	Symbol for Unit
length*	metre	m
mass*	kilogram	kg
time*	second	s
electric current*	ampere	A
thermodynamic*	kelvin	K
amount of substance*	mole	mol
luminous intensity*	candela	cd
plane angle+	radian	rad
solid angle+	steradian	sr
frequency	hertz	Hz
energy	joule	J
force	newton	N
pressure	pascal	Pa
power	watt	W
electric charge	coulomb	C
electric potential	volt	V
electric resistance	ohm	Ω
electric conductance	siemens	S
electric capacitance	farad	F
magnetic flux	weber	Wb
inductance	henry	H
magnetic flux density	tesla	T
luminous flux	lumen	lm
illuminance	lux	lx
activity (of radioactive source)	becquerel	Bq
absorbed dose (of ionising radiation)	gray	Gy

* SI Base Unit

+ Supplementary Unit

Table 9: The more commonly appearing quantities, their SI units, and alternatives in current use

Quantity	SI	Others in use and their abbreviations
length	m	centimetre(cm), parsec(pc), astronomical unit(AU)
mass	kg	gram(g), solar mass(Mo)
time	s	minute(m), hour(h), day(d), year(yr), Julian date(JD)
velocity	m/s	km/s
plane angle	rad	degree, arcminute, arcsecond
solid angle	sr	square degree, square arcsecond
wavelength	m	Angstrom, micron, centimetre
energy	J	erg
force	N	dyne
pressure	Pa	dyne/cm**2
density	kg/m**3	g/cm**3
power	W	erg/s
flux	W/m**2	erg/(cm**2*s)
luminous flux	lm	erg/(cm**2*s*sr), magnitude(mag)
spectral-flux density	Jy	erg/(cm**2*s*keV), erg/(cm**2*s*Ang)
surface brightness	Jy/sr	magnitudes/arcsecond**2
magnetic-flux density	T	gauss

6 History

For the storage of information related to the genealogy and processing history of a data frame, the Birmingham/Asterix design and subroutine-interface library has been adopted. Therefore the structure is named [**HISTORY**] and has TYPE <**HISTORY**>. (See Asterix Programmer Note 86_008.) A [**HISTORY**] structure describes the object in which it appears at the top level; it never describes anything at higher levels. This usually means that one history applies to one main data array.

History is optional and is under the control of applications—only they know whether they have done anything important to a file, and which parameters are important, though the user will always have the opportunity to add commentary via various patching and notepad utilities. There should be a logical parameter in applications to enable/disable history recording, which is normally defaulted in the interface file to disabled. Applications which do not modify or create structures containing [**HISTORY**] (*e.g.* <**NDF**>) do not need to update it — a display application, for example, would not have to write a history record.

History records must not be regarded by applications as machine readable—they must not be used to specify parameters or to control processing, even to re-enact automatically a processing sequence. They are present to assist the user: "What did I do to these data?", "Did I flat-field this image?".

History records should be brief.

Details are given in Section 10.6.

7 Bad-Pixel Methods

Starlink standard data formats support two methods of handling bad data: *magic value* (which flags specified pixels as undefined) and *data quality* (a more general mechanism, which may be used to indicate any attribute of selected pixels, including "badness"). Magic value is simple and efficient. Data quality is flexible and preserves the original data.

7.1 Quality

To flag a data value as "bad", an associated *data-quality* value can be used. This is an array of 8-bit positive integers, one per element of the data array with which it is associated (a single value, applying to all elements of the data array, is also possible, but this will rarely be useful), whose bits describe, in various ways, attributes of the data value concerned. The recommended way to use data quality is to regard the 8 bits as eight independent logical masks, one mask per attribute.

As its name implies, data-quality is a qualitative description of the data value. It is frequently used to flag bad pixels, but is also useful for "good" attributes, *e.g.* which regions of a picture constitute the sky sample. It is not in any sense an error estimate (though groups of bits might be used to convey some numerical meaning); it finds application in circumstances where an error estimate is not meaningful. Here are some examples of how data quality might be used:

- In an image where the intensities of some pixels have been digitally truncated, there is only a lower limit to the actual incident intensity; the upper limit is unbounded. Data-quality could be used to flag this condition, and application programs could then decide whether to use the pixel value or to treat it as missing.
- Data quality is useful where a pixel has an accurate intensity, but has to be interpreted in a different way from other pixels. The case of pixels affected by fiducial marks (*e.g.* reseaux) is a common example of this. For most parts of the processing, such pixels must be excluded. However, in an application which locates the fiducial marks themselves, they would clearly be crucial.
- Where parts of a picture are vignetted, data-quality allows these regions to be ignored when appropriate (at the discretion of the user, for example) without losing what information they contain.

Sometimes a simple true/false mask is not enough. In such cases it is possible to use combinations of bits to indicate both the presence of the condition and to what subclass of that condition the pixel belongs. For example, a group of three data quality bits could be used not only to flag saturation but also to grade the degree of saturation, on a scale of 1–7.

Clearly, not all values stored in the data system will have associated data-quality; that would be unnecessary and quite wasteful of resources. Normally, data-quality values are associated with basic observational or measured data.

7.2 Magic or Undefined Value

The alternative method for handling bad pixels is the so-called *magic value* method, where a pixel is assigned a special flag when it has an undefined value—it corresponds to a dead element in a CCD chip, for example, or is the result of division by zero. This terminology should not be confused with the HDS "undefined state", where a data object exists, but has no value(s) assigned to it. In this document "undefined" means "having a magic value", unless explicitly stated. An undefined pixel will always be bad, unless repaired in some fashion, and so the data-quality technique is not applicable.

The method is efficient on space: it can always be applied without increasing the data-storage requirement because the flag or magic value replaces the unwanted data value. (For applications where it is important to retain pixel values, or where there is a *degree* of badness, **data quality** should be used.) The method enables an application to discover whether a given pixel is bad as soon as it is accessed.

Alternative techniques, based on a list of bad pixels, would be less efficient, because the list would have to be searched repeatedly to see whether given pixels are bad. Such methods would be especially inefficient if large areas of pixels were undefined.

Once a bad pixel has been detected, the application can take appropriate action – flagging the corresponding output pixel as bad, or attempting a repair, perhaps *via* a choice of interpolation methods.

The HDS undefined state must not be used to indicate bad pixels. If an application finds a data-object in this state, it must report an error, so that the malfunctioning application which created the object can be identified and corrected. The error is fatal.

7.3 Implementation

General-purpose applications, like those in the KAPPA package, should support both magic-value and quality arrays. It will usually be best to look only for the magic-value case in the scientific algorithm part of the code, having dealt with any data-quality information in a preliminary pass which converts flagged pixels into magic-value ones.

The groups of true/false logical flags involved in the data quality mechanism are stored as integer values. We picture these integers as having conventional binary encoding, and adopt the convention that 1≡true. Specific VAX representations and conventions are **not** followed and are not in any way involved with the discussion.

7.3.1 Data Quality

Quality is an 8-bit value associated with each datum, and is stored as an unsigned byte. A value of zero (*i.e.* all quality flags set 0≡false) implies a "ordinary" value which can be accepted at face value by application programs.

General-purpose applications

In general-purpose applications, the data-quality values are regarded as a set of 8 independent masks, each of which is 1 bit deep. Whether a given pixel is to be included in the processing or not (*i.e.* whether it "bad") is determined by comparing its quality value with a bit pattern stored in a <**_UBYTE**> data object [**BADBITS**] within the <**QUALITY**> structure. The following logical expression is evaluated:

$$\mathrm{BAD} = \mathrm{QUALITY}\,\hat{}\,\mathrm{BADBITS}$$

where ^ is the logical AND operation.

Note that if a [**BADBITS**] mask is zero (*i.e.* all false), the corresponding data-quality mask is ignored. This can be used to turn off all 8 data-quality masks and allow inspection or processing of the pixels whatever their status. For a single bit, the above expression has the following truth table:

[**QUALITY**] bit	[**BADBITS**] bit	BAD
0	0	0
0	1	0
1	0	0
1	1	1

and the overall logical value of BAD is the OR of the results for all eight bits—just one of which has to be TRUE to make the resulting pixel bad.

An example may clarify this. Assume [**BADBITS**] is 01001010 (where the bits of the binary number are written with the most significant at the left, and are numbered from the right beginning with zero). For this [**BADBITS**] value, a pixel with a [**QUALITY**] value of 10100100 is interpreted as non-bad, because bits 2, 5 and 7, which are set in the data-quality value, are not set in [**BADBITS**]. However, a [**QUALITY**] value of 10100110 generates a bad value because bit 1, which is set in the data-quality value, is also set in [**BADBITS**]. If data object [**BADBITS**] is not present its value is assumed to be to be 00000000, and general-purpose applications will accept as "good" any pixel, irrespective of the corresponding data-quality value.

The rules and conventions for the processing of data-quality values and their associated data, taking into account the possible presence of undefined values, are as follows.

Rules —

- Undefined pixels stay bad after processing.
- Undefined pixels generated during the processing (other than through data quality), *e.g.* logarithm of a negative data value, are propagated to the output data value.
- If processing would or might have changed the value of a pixel, had the pixel not been marked as bad through data quality, then it must propagate an undefined pixel. The input quality is propagated. In applications where data value will not have been changed as a result of the processing, the application is permitted either (1) to propagate the original data value and quality or (2) to propagate an undefined pixel.

Conventions —

- When there is more than one input data array (*cf.* Section 9), the *input* or *original* data quality is deemed to be that associated with the *principal* data array.
  However, in some cases it is hard to identify a principal data array, or the principal data array does not have quality and one or more of the others does. Therefore, what is best depends on the nature of the application. For example, in the computation of the statistics of corresponding pixels from each of a series of pictures, to produce mean or standard-deviation arrays, it is vital to exclude **all** bad values from the calculations. A related problem is what the quality of output data arrays should be, and here again programmers must make case-by-case judgements.
- If an undefined pixel is generated during the processing, the data quality of the output data value is nonetheless the same as the input data quality, just as if a good pixel had been generated. (Although a pixel is undefined, you may still need to know, for example, that the pixel was a part of fiducial mark.)
- The original data and quality values remain unchanged.

If a [**QUALITY**] array is present it is assumed that it is to be used to define bad pixels unless:

- there is a parameter which overrides the default;
- the bit pattern of [**BADBITS**] is 00000000, or
- [**BADBITS**] is omitted from the <**QUALITY**> structure.

If [**QUALITY**] is not present the magic-value method is assumed.

There is no one ideal way of handling data quality in general-purpose routines. Methods will evolve as experience with real applications and data is gained. The main considerations are:

- Disc space and virtual memory—use of full-size work arrays may be unacceptable for large data frames.
- Speed—checks for magic value and data quality in pixel-by-pixel processing loops should be kept to a minimum. For example, the program could first determine whether bad pixels and data quality are present or not, and then call different processing routines for the two cases.

Table 10: Magic values for bad pixels

Data TYPE	Value	Hexadecimal pattern
<**_BYTE**>	−128	80
<**_UBYTE**>	255	FF
<**_WORD**>	−32768	8000
<**_UWORD**>	65535	FFFF
<**_INTEGER**>	−2147483648	80000000
<**_REAL**>	−1.7014117E+38	FFFFFFFF
<**_DOUBLE**>	−1.701411834604923D+38	FFFFFFFFFFFFFFFF

Specialist applications

Applications can be as sophisticated and specialised as they like in their use of data quality, and are at liberty to assign specific meanings to values of data quality, *e.g.* a fiducial mark, vignetting, saturation. The details of how data-quality information is encoded within the 8 bits are specific to each kind of data source and specialist package. A description of how quality will be interpreted must be given in the documentation for each package that uses the technique. However, it is possible to identify some general features of data-quality processing.

Each data-quality value can be regarded as a set of bit groups, each containing one or more bits. The recommended approach is to use single bits, each with an independent meaning, to form eight 1-bit deep logical masks. However, it is also permissible to take several bits (which ought to be contiguous) and interpret them as a positive integer. Single bit fields are used to contain a flag (1 = .TRUE., 0 = .FALSE.) for some feature (*e.g.* "pixel in fiducial"). Multiple-bit fields are used to contain code numbers or degree of quality.

It is envisaged that most manipulation of data-quality values will be done quite transparently by those applications which know how to use them to advantage, without the user being aware of the mechanism. However, it is expected that there will be some cases where users will want to manipulate data quality explicitly, and there will be various data-quality editing applications, often using graphics or image displays. For example, there will be instances where the user wishes to view a picture on a display and select which pixels are to be temporarily flagged as "wrong", rather than trust some automatic algorithm.

Since the data quality codes are stored separately from the actual data, data-quality editing will normally be a reversible process, leaving the data values themselves untouched.

(*n.b.* The implementation of data quality is largely unchanged from the Wright-Giddings proposal.)

7.3.2 Magic Values

For each of HDS's primitive TYPEs, the magic-value method uses the values given in Table 10. Alert readers will note that these are the same as the bad values used by HDS.

Use of "undefined data" flags must be restricted to three operations: (i) setting a datum to "undefined", (ii) testing whether a datum is in the undefined state, and (iii) replacing an undefined datum with a valid value (using an assignment statement). Arithmetic operations on undefined data values are **banned**. Magic values are applicable to both scalar and vector data objects. There are some exceptions and these are individually noted.

For efficiency, pixel values are tested inline for equality with the magic value of the appropriate type. However, the numerical values given above must not be written explicitly in the code; instead, variables called `VAL__BAD<T>`, where `<T>` is the one or two-letter type code (*e.g.* see SUN/7), should be used. These variables are specified via an INCLUDE file with logical name BAD_PAR. Here is a trivial example, which computes the mean of a one-dimensional REAL array. (*n.b.* Actual applications would include comments and defences against rounding errors, excluded for brevity here.)

```
INTEGER I, N, NPIX
PARAMETER (NPIX = 100)

REAL DATA(NPIX), SUM, MEAN

INCLUDE 'BAD_PAR'

SUM = 0.0
N = 0
DO I = 1, NPIX
   IF ( DATA( I ) .NE. VAL__BADR ) THEN
      SUM = SUM + DATA( I )
      N = N + 1
   END IF
END DO

IF ( N .EQ. 0 ) THEN
   MEAN = VAL__BADR
ELSE
   MEAN = SUM/ REAL( N )
END
```

Note that only valid pixels are counted and summed.

For reasons of efficiency of processor time and work space, and to permit easier portability and adaptation of general-purpose subroutine libraries, a flag called [**BAD_PIXEL**] may be provided within a structure to denote whether undefined pixels are present. Only if it is present and set to `.FALSE.` will it be permissible to bypass magic-value testing. Thus, many packages will support two sets of algorithmic subroutines; one which tests magic values, and one which does not.

8 Extensions

The flexible and general-purpose character of the structures described in this document stems from their simplicity. This was a conscious design decision, the more traditional "we've thought of everything" approach having been rejected for reasons given earlier. By keeping the structures

simple, it is possible to be reasonably precise about the processing rules, and this will enable users to mix applications from different packages in their processing schemes.

However, the standard structures do not immediately cater for package- or instrument-specific objects and their processing, and indeed show very little evidence of their planned astronomical rôle, which is realised through the use of *extensions*.

Extension information can be stored only in specially reserved places within certain standard data structures (including the NDF). These places comprise optional structures of NAME [**MORE**] TYPE <**EXT**>, which contain, in turn, the extensions proper, usually of TYPE <**EXT**>. The NAMEs and TYPEs of the extensions, self-contained assemblies of related information peculiar to an application package or data source, must be registered with the Starlink Head of Application to avoid clashes. (Which [**MORE**] they go in has also to be specified.) Programmers should select names which are sensible, descriptive and related to the extension and the project concerned. Simple generic names should be avoided in case they are needed later for Starlink general-purpose packages, *e.g.* [**HEADERS**], [**PHOTOMETRY**].

Starlink will, in due course, define standard representations of time and celestial positions (using the rules of Section 13). They will be implemented as extensions, not additions to the standard structures. It has not yet been decided whether applications in the KAPPA package will process these standard extensions, or whether a new package, of basic astronomy applications, will be written.

9 Propagation of Data Objects

Many applications take one or more *input* objects and generate one or more *output* objects. In such cases, applications have to make decisions about what parts of the input objects are to be propagated to the outputs, guided by the following rules:

- General-purpose applications will copy extensions (*cf.* Section 8) to the same location within an output structure provided their parent structure has been propagated; specialist applications are, of course, allowed to process individual items within the extensions they understand.
- Items whose properties are well understood in the context of a given application may be processed by that application.
- Items whose properties are not well defined in the context of a given application must **not** be propagated by that application. Thus, items or structures rendered meaningless, wrong or even dubious by the processing must not be propagated. The *ad hoc* approach to doubtful cases is strongly discouraged.
- "Rogue" objects—anything outside the Starlink standard—should be ignored.
- Applications are not allowed to pollute standard structures with rogue objects; all application-specific items must reside in registered extensions, within one of the [**MORE**] structures.

By stipulating that rogue objects must be ignored, we avoid the problem of what should happen if a component of the same name already exists in the output structure. Also, not having to

worry about rogue objects will greatly simplify the checking and revision of application code that would follow any revision or extension of the standard structure definitions in the future.

In cases where applications access more than one composite data structure to generate a single output structure, *e.g.* arithmetic of two data arrays, there are additional problems about the propagation of certain data objects. These are mainly descriptive items *e.g.* [**HISTORY**], [**LABEL**]. Should one, all or none of these data objects be propagated? If the third rule above is followed rigorously the answer is none, but many useful and relevant data objects would then be lost. To assist in deciding whether they can be retained, the concept of a dominant or *principal data array* is introduced, whose various associated items will be the ones propagated. By convention, the application's interface file will be arranged so that, of the parameters specifying data arrays, the principal array will be the first in sequence. In many cases, it will be necessary to advise the user which items have been propagated and which have been lost; if the result is unsuitable, the application can then be re-run with the parameters specified in a different order, or other programs can be run to fine-tune the result—by replacing a label, for example.

The specific rules for HISTORY are as follows.

If there is a principal array, its history should be propagated with a record appended for the latest operation, which could include the name and title of the secondary data arrays. If the data arrays are equally important, it is probably best to create a new [**HISTORY**] structure, again with the names/titles of its parents. However, for cases where this option will not suffice, a special HDS editor application will be provided for grafting in history records taken from existing <**HISTORY**> structures.

10 Low-Level Structures

In this section, we begin by describing a number of *low-level* structures, the basic building blocks which can be used to build NDFs and other *composite* structures.

10.1 <POLYNOMIAL> Structure

The <**POLYNOMIAL**> structure is used for storing the coefficients of an n-dimensional ordinary or Chebyshev polynomial, or the coefficients of a cubic B-spline. The value of the [**VARIANT**] component defines which of these types of polynomial is stored. Evaluation of the polynomial yields a <**float**> scalar result.

In the following descriptions NAXIS is the dimensionality of the [**DATA_ARRAY**] and therefore the dimensionality of the space over which the polynomial function is defined.

10.1.1 [VARIANT] = 'SIMPLE'

[DATA_ARRAY] This is mandatory. The scalar (VALUE) which results from the evaluation of the polynomial inherits the equivalent primitive TYPE of the polynomial coefficients.

The [**DATA_ARRAY**] is the $NAXIS$-dimensional array containing the $NAXIS$-dimensional polynomial coefficients. The function is evaluated as:

$$\mathrm{VALUE} = \sum \mathrm{DATA\_ARRAY}(I_1, I_2, \ldots, I_{\mathrm{NAXIS}}) \prod_{j=1}^{\mathrm{NAXIS}} \mathrm{AXIS}_j^{I_j - 1}$$

Table 11: Contents of the <**POLYNOMIAL**> Structure, [**VARIANT**] = 'SIMPLE'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SIMPLE'
[**DATA_ARRAY**]	<**farray**>	coefficients
[**VARIANCE**]	<**farray**>	variance of the coefficients

summed over all elements of the DATA_ARRAY.

Here, AXIS_n is the co-ordinate value along axis n, and NAXIS_n is the dimension of axis n.

[VARIANCE] This optional component is used to describe the variance of the errors associated with the polynomial coefficients given by the [**DATA_ARRAY**] component, and it must be an array of the same dimension as [**DATA_ARRAY**].

10.1.2 [VARIANT] = 'CHEBYSHEV'

Table 12: Contents of the <**POLYNOMIAL**> Structure, [**VARIANT**] = 'CHEBYSHEV'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'CHEBYSHEV'
[**DATA_ARRAY**]	<**farray**>	Chebyshev coefficients
[**VARIANCE**]	<**farray**>	variance of the coefficients
[**TMIN**(NAXIS)]	<**float**>	lower bound of co-ordinate range
[**TMAX**(NAXIS)]	<**float**>	upper bound of co-ordinate range

[DATA_ARRAY] This is mandatory. The interpretation The scalar (VALUE) which results from the evaluation of the polynomial inherits the equivalent primitive TYPE of the polynomial coefficients.

The [**DATA_ARRAY**] contains the NAXIS-dimensional array of Chebyshev-polynomial coefficients. The function is evaluated as:

$$\text{VALUE} = \sum \text{DATA\_ARRAY}(I_1, I_2, \ldots, I_{\text{NAXIS}}) \prod_{j=1}^{\text{NAXIS}} T(I_{j-1}, S_j)$$

summed over all elements of the DATA_ARRAY, where $T(m, x)$ is the Chebyshev polynomial of order m evaluated at x (in the range -1 to $+1$), and

$$S_n = (2 * \text{AXIS}_n - (\text{TMAX}(n) + \text{TMIN}(n))/(\text{TMAX}(n) - \text{TMIN}(n))$$

If the polynomial is evaluated outside this interval, the result is bad, and set to the magic value.

[VARIANCE] This optional component is used to describe the variance of the errors associated with the Chebyshev polynomial coefficients given by the [**DATA_ARRAY**] component, and it must be an array of the same dimension as [**DATA_ARRAY**].

[TMIN(*n*)], [TMAX(*n*)] These are the minimum and maximum values along AXIS number n. They represent the bounds of the normalised co-ordinate range. They are mandatory.

10.1.3 [VARIANT] = 'BSPLINE'

Table 13: Contents of the <**POLYNOMIAL**> Structure, [**VARIANT**] = 'BSPLINE'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'BSPLINE'
[**DATA_ARRAY**]	<**farray**>	coefficients
[**VARIANCE**]	<**farray**>	variance of the coefficients
[**SCALE**]	<**float**>	normalisation
[**KNOTS**]	<**farray**>	co-ordinates of the knots

The [**DATA_ARRAY**] contains the NAXIS-dimensional array of cubic-spline coefficients. Its dimension along an axis is equal to the total number of knots along that axis less four. The function is evaluated as:

$$\text{VALUE} = 1/\text{SCALE} \sum \text{DATA\_ARRAY}(I_1, I_2, \ldots, I_{\text{NAXIS}}) \prod_{j=1}^{\text{NAXIS}} B_j(\text{AXIS}_j)$$

where $B_j(\text{AXIS}_j)$ is the normalised cubic B-spline along the jth axis with the knots recorded in [**KNOTS**], and SCALE is the value of component [**SCALE**].

[VARIANCE] This optional component is used to describe the variance of the errors associated with the spline coefficients given by the [**DATA_ARRAY**] component, and it must be an array of the same dimension as [**DATA_ARRAY**].

[KNOTS] A vector of knot co-ordinates, starting with those along the first dimension, followed by any second dimension, and so on. It has a dimension that is the sum of the number of knots along all axes.

[SCALE] The normalisation factor to convert the evaluated spline into the fitted value.

10.2 <ARRAY>Structure

We will now describe the various structures for arrays of numbers introduced in Section 3.3. An <**ARRAY**> structure has a number of variants.

Table 14: Components of <**ARRAY**> Structure, [**VARIANT**] = 'SIMPLE'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SIMPLE'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DATA**]	<**narray**>	actual value at every pixel

10.2.1 [VARIANT] = 'SIMPLE'

[ORIGIN] The pixel origin in each dimension. If this object is not present, each origin is assumed to be 1. Note that this ability, though present in FORTRAN77, is absent from HDS *per se*.

This information may be communicated to access routines via the upper and lower bounds on each array dimension, so that the Fortran dimensions would be

```
ARRAY(ORIGIN(1):DIMS(1)+ORIGIN(1)-1, ORIGIN(2):DIMS(2)+ORIGIN(2)-1, ...).
```

[**ORIGIN**] overcomes a number of quite serious problems, including the old chestnut "does the first pixel start at (1,1) or (0,0)?", because the origin is specified explicitly. Also negative pixel indices become available, and therefore an application is freed from having every array co-located at the first pixel, and it may extend arrays in any direction. This property is particularly useful for storing data which require negative indices for a natural representation, such as Fourier-transform data, where zero often needs to be in the centre of the array; or image data after (say) rotation about any point. Storing a pixel origin also enable applications to cut out parts of a data array yet maintain the array's coordinate system (it can be re-combined with the original if necessary). Since the coordinate system is preserved, all the axis scaling is also preserved. This means, for instance, that applications do not have to re-compute the polynomial coefficients in an <**AXIS**> structure when the array size is changed.

NAXIS The number of axes in the array being represented.

[DATA] This mandatory item is the NAXIS-dimensional array of numbers.

10.2.2 [VARIANT] = 'SCALED'

A scaled <**ARRAY**> is sometimes referred to as *block floating point*. When dealing with large-format data, disk space is a major consideration; significant space may be saved by storing data in a 16-bit form via the block floating-point format. Scaling may also be used as a means of implementing global scalar changes in the array's values without the need to change the array, or to move to a more expensive storage type (*e.g.* to <**float**> from <**_WORD**>) in order to work with very small or very large values.

General-purpose applications will process the scaled variant. However, users may notice extra delays while the array is converted to <**narray**> for processing and rescaled for storage at the end of each application. In such cases, they may prefer the alternative strategy of first running a utility application to convert the data to primitive floating point in a new container file, then

Table 15: Components of <**ARRAY**> Structure, [**VARIANT**] = 'SCALED'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SCALED'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DATA**]	<**narray**>	scaled numeric value at every pixel
[**SCALE**]	<**numeric**>	scale factor
[**ZERO**]	<**numeric**>	zero point

performing their processing, and finally running another utility to convert the data back to the scaled-array form.

The numeric value of a pixel is:

```
ZERO + DATA element * SCALE.
```

NAXIS The number of axes in the array being represented.

[ORIGIN] The pixel origin in each dimension. If this object is not present, each origin is assumed to be 1.

[DATA] The array stored in scaled (block-floating-point) format. Mandatory.

[SCALE] A positive scale factor used to convert the element of [**DATA**] to its unscaled <**numeric**> form. If it is not present, or is negative, or is bad, an error condition results. It is evaluated as follows:

```
SCALE = (MAX - MIN)/(TMAX<T> - TMIN<T>)
```

where MAX, MIN are the maximum and minimum valid (*i.e.* not bad) data values respectively, and TMAX<T>, TMIN<T> are the largest and smallest valid values for the primitive type of [**DATA**], represented by token <T>.

[ZERO] The zero point used to convert the element of [**DATA**] to its unscaled <**numeric**> form. If it is not present, or is undefined, a value of zero is assumed. It is evaluated as follows:

```
ZERO = TMIN<T> - SCALE * MIN
```

The equivalent primitive type is the TYPE of [**ZERO**] or, if [**ZERO**] is absent, the TYPE of [**SCALE**]. If both [**ZERO**] and [**SCALE**] are missing the equivalent <**numeric**> array defaults to TYPE <**_REAL**>.

Table 16: Components of <**ARRAY**> Structure, [**VARIANT**] = 'SPACED'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SPACED'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DIMENSIONS**(NAXIS)]	<**integer**>	dimensions of equivalent numeric array
[**SCALE**(NAXIS)]	<**numeric**>	scale factors
[**BASE**(NAXIS)]	<**numeric**>	zero points

10.2.3 [VARIANT] = 'SPACED'

A spaced <**ARRAY**> is one where the data values vary linearly with array index. An example where this variant could be used is axis data, which are often equally spaced.

General-purpose applications will handle these structures. However, extra time will be expended (though probably much less than for a scaled <**ARRAY**>) whilst conversion to and from a <**narray**> is performed. They do not contain magic-value bad pixels.

The value of an element of the equivalent <**numeric**> array would be:

$$\text{VALUE} = \sum_{i=1}^{\text{NAXIS}} (\text{BASE}(i) + (k_i - 1) * \text{SCALE}(i))$$

where k_i is the element number of the i^{th} dimension.

NAXIS The number of axes in the array being represented.

[ORIGIN] The pixel origin in each dimension. If this object is not present, each origin is assumed to be 1.

[SCALE] The scale factor for each dimension used to convert the corresponding array index of [**DATA**] to its <**numeric**> form. If it is not present, or is undefined, a value of 1 is assumed for each dimension.

[BASE] The zero point for each dimension used to convert the corresponding array index of [**DATA**] to its <**numeric**> form. If it is not present, or is undefined, a value of 0 is assumed for each dimension.

[DIMENSIONS] The axis dimensions of the full array. Mandatory. The equivalent primitive type is the TYPE of [**BASE**], or if [**BASE**] is absent, the TYPE of [**SCALE**]. If both [**BASE**] and [**SCALE**] are missing, the equivalent <**numeric**> array defaults to TYPE <**_REAL**>.

10.2.4 [VARIANT] = 'SPARSE'

A sparse <**ARRAY**> structure is used for storing the values of an array where most of the pixels are equal in value (zero perhaps). The structure is composed of a list of array indices for the specified "non-grey" elements, and their corresponding data values. The "grey" value adopted for the majority of the array elements is also supplied.

Table 17: Components of <**ARRAY**> Structure, [**VARIANT**] = 'SPARSE'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'SPARSE'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DATA**(NDATA)]	<**numeric**>	data values for pixel subset
[**LIST**(NAXIS,NDATA)]	<**integer**>	list of pixel array indices
[**DIMENSIONS**(NAXIS)]	<**integer**>	dimension of equivalent <**numeric**> array
[**GREY**]	<**numeric**>	grey value

NAXIS The number of axes in the array being represented.

[ORIGIN] The pixel origin in each dimension. If this object is not present, each origin is assumed to be 1.

NDATA This is the number of non-grey array elements.

[DATA] These are the data values for significant array elements. Mandatory.

[LIST] These are the array indices for each significant array value. Mandatory.

[DIMENSIONS] This array contains the axis dimensions of the full data array. It is used to generate the equivalent <**narray**>. Mandatory.

[GREY] This is the value used to fill the array elements not referenced in the [**LIST**] component. It should have the same primitive TYPE as [**DATA**]. If absent, it is assumed to be the bad-pixel magic value (for [**DATA**]'s primitive type) since the pixel's value is undefined.

The equivalent primitive type is the TYPE of [**DATA**].

10.2.5 [VARIANT] = 'POLYNOMIAL'

The array is represented by an n-dimensional polynomial. (This variant does not conform to the guideline about variants since conversion of an array of numbers to a polynomial requires additional information. However, for clarity and because the reverse operation is possible—polynomial to an array of numbers—a variant of <**ARRAY**> has been used.)

NAXIS The number of axes in the array being represented.

[ORIGIN] The pixel origin in each dimension. If this object is not present, each origin is assumed to be 1. This avoids recomputing the polynomial coefficients if a subsection of a larger array is created.

[DATA] The NAXIS-dimensional polynomial representing the array of numbers. Mandatory. The equivalent primitive type has the same TYPE as the polynomial coefficients. To generate the equivalent primitive array, the polynomial is evaluated at pixel centres, which start at 0.5 for the first pixel given a uniform bin.

Table 18: Components of <**ARRAY**> Structure, [**VARIANT**] = 'POLYNOMIAL'

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	'POLYNOMIAL'
[**ORIGIN**(NAXIS)]	<**integer**>	origin of the data array
[**DATA**]	<**POLYNOMIAL**>	polynomial representing array
[**DIMENSIONS**(NAXIS)]	<**integer**>	dimension of equivalent <**numeric**> array

Table 19: Components of <**COMPLEX_ARRAY**> Structure

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	registered variant
[**REAL**]	<**p_array**>	real-component data values
[**IMAGINARY**]	<**p_array**>	imaginary-component data values

[DIMENSIONS] This array contains the axis dimensions of the full data array. It is used to generate the equivalent <**narray**>. Mandatory.

The 'SPACED' variant may appear merely to be a degenerate form of the 'POLYNOMIAL' variant. However, there are reasons for having a 'SPACED' variant:

- the case where the equivalent primitive type is <**integer**> is common, whereas the polynomial variant only produces <**float**> results.
- The spaced <**ARRAY**> uses the pixel indices to generate the values, whereas a polynomial array uses the pixel co-ordinates. Pixel indices are integers, starting at 1; pixel co-ordinates are real numbers, which start at 0.5 for uniformly-sized bins.
- The spaced <**ARRAY**> format will be easier to understand in a structure listing.

To reiterate, general-purpose applications will be able to process <**ARRAY**> structures, though their performance will be degraded when the [**VARIANT**] is not 'SIMPLE' because of the extra processing required. These options do not create a significant overhead for the normal case, the base variant. When the [**VARIANT**] is not 'SIMPLE', users will probably be warned that their data are in a special format, be given the option to abort the application, and be directed to documentation for applications better suited to the data.

10.3 <COMPLEX_ARRAY> Structure

HDS does not provide a <**_COMPLEX**> primitive type, and therefore if complex data are to be stored a <**COMPLEX_ARRAY**> structure, defined in Table 19 must be used.

Currently, the [[VARIANT]] can only be 'SIMPLE'.

Table 20: Components of <**AXIS**> structure element

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	registered variant
[**DATA_ARRAY**]	<**p_array**>	axis value at each pixel
[**LABEL**]	<**_CHAR**>	axis label
[**UNITS**]	<**_CHAR**>	axis units
[**VARIANCE**]	<**s_array**>	axis variance in [**DATA_ARRAY**]
[**NORMALISED**]	<**_LOGICAL**>	true if the data have been normalised
[**WIDTH**]	<**s_array**>	bin width of each pixel
[**MORE**]	<**EXT**>	extension structure

The dimensions of [**REAL**] and [**IMAGINARY**] must be identical. Both components are mandatory. The pixel origin will be taken from [**REAL**] component's structure, *i.e.* [**REAL.ORIGIN**].

If a pixel is undefined, the magic value should be assigned to both corresponding elements of the real and imaginary arrays. However, should the application encounter the case where only one element is flagged, the whole pixel should be regarded as undefined.

The [**ORIGIN**] components should match, but only [**REAL.ORIGIN**] is used.

10.4 <AXIS> Structure

This structure has TYPE <**AXIS**> and is used to store information describing the size and spacing of the pixels in a multi-dimensional data array. In use, <**AXIS**> structures are arranged as elements of a 1-dimensional array (of structures). The number of elements (structures) in this array is equal to the dimensionality (number of axes) of the data array being described, so that each <**AXIS**> structure relates to a single data dimension. The following description applies to one element of that multi-dimensional structure.

[VARIANT] Currently, this can only be `'SIMPLE'`.

[DATA_ARRAY] This mandatory component, which can be the only item within the <**AXIS**> structure, is used to provide co-ordinates along the axis. Each value corresponds to the *centre* of a pixel. For an n-dimensional data array, with no axis calibration, and equally spaced data, the Starlink standard is for the n^{th} pixel centre to be at $(n-1).5$ in each axis, *e.g.* for display purposes (*cf.* SSN/22). However, in some applications the pixel index is adequate and clearer. Programmers should take care to use the appropriate scheme.

The co-ordinates are given as a vector with length corresponding to that of the corresponding dimension of the data array. In practice, it will often be the polynomial or spaced variant of TYPE <**ARRAY**>.

[LABEL] This is a textual description of the co-ordinate type. It may contain the "control" codes that are used by graphics packages to produce special characters—backslashes *etc*.

Table 21: Components of <**QUALITY**> structure

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	registered variant
[**QUALITY**]	<**p_array**>	quality value of each pixel
[**BADBITS**]	<**_UBYTE**>	mask of allowed bad bits in quality

[UNITS] This is a textual description of the co-ordinate units.

[VARIANCE] This component is used to describe the variance of the errors associated with co-ordinate values given by the [**DATA_ARRAY**] component. It must either be an array of the same dimension as [**DATA_ARRAY**] or, should a single variance apply to all elements of [**DATA_ARRAY**], a scalar.

[NORMALISED] If true, the data have been normalised to the pixel size (*i.e.* divided by the pixel width along this axis). In other words if the data are normalised the data values reduce as the pixels are stretched and *vice versa*. Optional. If absent, it defaults to false.

[WIDTH] The width or extent of each bin for irregularly spaced and/or overlapping data. If it is an array, it must have the same dimensions as the [**DATA_ARRAY**]. The scalar form applies to all the pixels and is intended for overlapping pixels. Optional. If absent, the extent of the n^{th} element is assumed to be

$$0.5\ (D_{n-1} + D_n) \quad \text{to} \quad 0.5\ (D_n + D_{n+1}),$$

where D_n is the n^{th} centre value given by [**DATA_ARRAY**]. The extreme values are twice the available half widths, *viz.*

$$D_n - 0.5 * \text{WIDTH} \quad \text{to} \quad D_n + 0.5 * \text{WIDTH}$$

for the n^{th} element.

[MORE] The extension in which application-specific axis-related information can be stored, *e.g.* whether the axis is cyclic, the type of axis: spectral, spatial, temporal *etc.*

10.5 <QUALITY> Structure

This structure is used for storing data-quality information.

[VARIANT] Currently can only be `'SIMPLE'`.

[QUALITY] The data-quality value(s). If it is scalar it applies to all the pixels. The actual or equivalent primitive type must be <**_UBYTE**>. Mandatory.

The rôle of data-quality and how it can be used to advantage were described earlier.

[BADBITS] A set of 8 one-bit flags which flag that the corresponding data quality bits indicate bad pixels; if omitted a default value of zero (*i.e.* data quality is ignored in deciding whether a pixel is bad or not) is assumed.

Table 22: Contents of the <**HISTORY**> Structure

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	registered variant
[**CREATED**]	<**_CHAR**>	creation date and time
[**EXTEND_SIZE**]	<**_INTEGER**>	increment number of records
[**CURRENT_RECORD**]	<**_INTEGER**>	record number being used
[**RECORDS**(*m*)]	<**HIST_REC**>	array of *m* history records

Table 23: Contents of the <**HIST_REC**> Structure

Component Name	TYPE	Brief Description
[**VARIANT**]	<**_CHAR**>	registered variant
[**DATE**]	<**_CHAR**>	creation date and time
[**COMMAND**]	<**_CHAR**>	application name and version
[**TEXT**(*m*)]	<**_CHAR**>	*m* lines of text

10.6 <HISTORY> Structure

This structure has NAME [**HISTORY**] and TYPE <**HISTORY**>. Its components are shown in Table 22.

[VARIANT] Currently, this can only be `'SIMPLE'`.

[CREATED] The date and time when the HISTORY structure was created, derived from the computer's clock. Mandatory. The format for dates and times is as following examples:

```
1988/AUG/07 23:59:59.999
2001-JAN-01 01:02:03.456
```

[EXTEND_SIZE] The number of history records to be created whenever the existing records have been filled. The default is 5. Initially 10 records are created. (The item [**INITIAL_SIZE**] in the original ASTERIX history system is redundant and has been removed.)

[CURRENT_RECORD] The number of the current history record. Mandatory.

[RECORDS] An array of <**HIST_REC**> structures containing the history records. Mandatory.

[VARIANT] Currently, this can only be `'SIMPLE'`.

[DATE] The date and time when the record was created, derived from the computer's clock. Mandatory.

[COMMAND] The name and version number of the application or command which modified or created the structure at or below the level in the hierarchy of this <**HISTORY**> object. Mandatory.

[TEXT] History text, which can span an unlimited number of records. It may contain commentary or details of input parameters. Parameters should be converted to text using environment calls.

Note there are currently no variants of the <**HISTORY**> and <**HIST_REC**> structures.

11 The Extensible n-Dimensional-Data Format

The most-common structure for data that are not instrument specific is what has become known as the *bulk-data frame*. To avoid confusion with the Interim Environment's BDF, the new Starlink standard for storing bulk-data frames is called the Extensible *n*-Dimensional-Data Format (NDF for short). It has no specific HDS NAME because a container file may have several <**NDF**> structures at a given level. It has an optional TYPE of <**NDF**> that will not be tested by general-purpose applications but is recommended to assist recognition by human readers of structure listings. NDFs may be structured recursively—see the polarimetry example below, for example.

It was not, in fact, possible to keep strictly to the rules in Section 13 when designing the NDF structure; compromises were necessary in order to allow old Asterix and Wright-Giddings-formatted data, of which there is a great deal, to be processed by the new general-purpose applications. The NDF structure comprises a title, a data array and its associated objects (a [**DATA_ARRAY**] structure in the Wright-Giddings terminology), axis information, history and one or more registered named objects containing application-specific components. Note that everything at the top level is intended to be under Starlink control, and although general-purpose applications will (for an initial period) tolerate non-standard components at this level, such *rogue objects* will not be processed beyond being copied to the same place within an output structure. This Starlink-components-only restriction, which does not preclude extensibility (done through the MORE objects), simplifies the job of applications, relieving them of the responsibility of keeping track of arbitrary numbers of extra objects. It is recommended that if an application detects the presence of a rogue object it should display a warning message, to alert the user to take some action (for example to run the appropriate format conversion utility).

[VARIANT] Specifies which sort of <**NDF**> structure. The variant must be one of the registered strings, of which only `'SIMPLE'` is currently available.

[TITLE] A title for the data which may be used to annotate plots and listings, and which will help identify the NDF. (A single line of text will obviously be too brief to describe the contents of a dataset in detail, but will be useful for display purposes.)

[DATA_ARRAY] This is the primary *n*-dimensional array of data values. It is the ONLY obligatory structure element. The [**DATA_ARRAY**] can be present in one of these forms:

(1) A <**narray**>. This primitive form, *i.e.* just a <**numeric**> array of numbers, is available to give compatibility with the old Wright-Giddings proposals. Although its

Table 24: Components of the Extensible *n*-Dimensional-Data structure

Component Name	TYPE	Brief Description
[VARIANT]	<**_CHAR**>	variant of the <**NDF**> type
[TITLE]	<**_CHAR**>	title of <**NDF**>
[DATA_ARRAY]	<**various**>	NAXIS-dimensional data array
[LABEL]	<**_CHAR**>	label describing the data array
[UNITS]	<**_CHAR**>	units of the data array
[VARIANCE]	<**s_array**>	variance of the data array
[BAD_PIXEL]	<**_LOGICAL**>	bad pixel flag
[QUALITY]	<**various**>	quality of the data array
[AXIS(NAXIS)]	<**AXIS**>	axis values, labels, units and errors
[HISTORY]	<**HISTORY**>	history structure
[MORE]	<**EXT**>	extension structure

use is discouraged for new applications, it is recommended that general-purpose applications propagate the <**narray**> format, as input, rather than convert to an <**ARRAY**> structure.

(2) A <**c_array**>.

[LABEL] This is a textual description for the kind of quantity stored in the **[DATA_ARRAY]** array.

[UNITS] This is a textual description for the units in which the data values are given. If more than one NDF is being processed, the various **[UNITS]** text may be tested for equality. Should they prove unequal, the application must inform the user, who then may have an opportunity to permit processing to continue; however, **[UNITS]** would **not** under these circumstances be propagated to the output NDF if any.

[VARIANCE] This is used to store the variance of the errors associated with **[DATA_ARRAY]**. It is used for computing symmetric error bars. The array dimensions must correspond to those of the **[DATA_ARRAY]** component. If all values in the data array have the same error, this can be represented by the scalar option. Other, more complex, forms of error representation (*e.g.* asymmetric errors) can be stored in specialised extensions, yet to be defined.

[BAD_PIXEL] If this is false, applications may assume that **[DATA_ARRAY]** and **[VARIANCE]** contain no magic-value pixels. If it is either true or absent, applications must either test for magic-value pixels or—if incapable of performing bad-pixel processing—give up.

[QUALITY] The data-quality values for the corresponding elements of **[DATA_ARRAY]**. Its TYPE is either <**narray**> or <**QUALITY**>. Note that the array can be stored in a sparse variant <**ARRAY**> structure; however, the dimensions of the sparse array must correspond to those of the **[DATA_ARRAY]** component, and the actual or equivalent primitive

type for the data-quality values must be <**_UBYTE**>. The <**narray**> option is to allow compatability with existing data in Wright-Giddings format; there was no [**BADBITS**] flag in the Wright-Giddings format, and so when such existing data are processed by general-purpose applications non-zero data quality will not be interpreted as bad pixels as would have occurred formerly.

NAXIS The dimensionality of the [**DATA_ARRAY**], and therefore the number of elements (structures) in the AXIS array (of structures).

[AXIS] This is an array of <**AXIS**> structures, where [**AXIS**(*n*)] corresponds to the n^{th} dimension of the [**DATA_ARRAY**]. If [**AXIS**] is not present the pixel index is used, starting from the associated value(s) of [**DATA_ARRAY.ORIGIN**], or 1 if the origin data object does not exist. If a simple pixel index is required, then the [**AXIS**] should be omitted from the <**NDF**>.

[MORE] This is a wrapper containing extensions; it is *not* itself an extension. The extensions and their components are outside the scope of the NDF definition, and they will be defined separately and in many cases will belong to specific applications packages. Each extension must have a unique NAME, by which it is recognised. Its TYPE may be any one of the Starlink-defined standard TYPEs, or may a new one defined according to rules in Section 13. Each extension (with the NAME, TYPE and variants) must be registered with the Starlink Head of Applications. Further NDFs may be located within these structures, and these may in turn contain extensions. To reduce the task of registration, and to minimise the risk of clashes, it is strongly recommended that one structure per application package be used rather than multiple minor items. It is also recommended that hierarchical structuring be used within extensions (rather than just 'flat' lists of components) so as to group related data objects, *e.g.* by processing or instrument.

Notes:

(1) Locating the data array.

General-purpose applications expecting an <**NDF**> structure should be prepared to process the data array of Wright-Giddings formats as well. Also, it should not matter in either case whether the name of the structure containing the data array or the name of the data array itself is supplied by the user. However, only when the name of the <**NDF**> structure is given can other data objects in the NDF be processed, because of the no-tree-walking rule. An outline algorithm to achieve the required functionality is:

```
Given name of object
Find its type
if (type not primitive) then
    if (type not <c_array>) then
        if (type not <NDF>)then
            issue warning but proceed
        endif
        look for [DATA_ARRAY]
        if ([DATA_ARRAY] not found) then
            No data processed
```

```
            Exit
        else
            Search for other required items
        endif
    endif
endif
Process [DATA_ARRAY].
```

(2) Accessing part of a [**DATA_ARRAY**]

Some general-purpose applications will need to be able to access subsets of a data array. The problem is twofold: first, the method of implementation needs to be specified, and second, the representation of each axis must be identified. An example is a general image-display routine which expects to be supplied a two-dimensional image but which is instead given a three-dimensional data cube. Such an application must have a means to select the whole or part of a slice from the cube. One method is simply to use two applications one after the other: first run MANIC (a KAPPA application) on the input data array to create a new dataset containing the required data; and second, run the required processing application on those extracted data. However, this means extra work for the user, and extra scratch space requirements, and in the case of frequently-used applications it will be more natural to provide the necessary 'slicing' capability directly. In these cases, applications will be able to exploit MANIC's component subroutines, which will first obtain the parameter values to specify the data subset required, and then extract the subset efficiently and store it in internal workspace ready for processing. Through the applications interface file, it will be possible to set up default parameter values tailored to the application concerned. When the selection of axes is being made (specifying in what direction the 2-D cut through the 3-D data cube is to be made, for example), the application should display to the user the axis labels (if present) to assist identification.

(3) Higher-level structures

Various specialised data objects and structures may be packaged around the NDF structure, using the NDF as a building block. One common requirement is for a series of related spectra or pictures; this could be implemented simply as a sequence of NDFs as follows:

```
name     special_type
  [name1] <NDF>
     ⋮         ⋮
  [name2] <NDF>
     ⋮         ⋮
  [name3] <NDF>
     ⋮         ⋮
```

Another approach would be to use an HDS array, each element of which is an NDF.

(4) Merging two or more <**NDF**> structures

The merging of history records has been discussed in Section 10.6, and the same approach is followed for other data objects within an <**NDF**>. Thus, cases are divided into (i)

Table 25: Example Polarimetry extension.

Component Name	TYPE	Brief Description
[STOKES_Q]	<**NDF**>	Stokes Q data objects
[STOKES_U]	<**NDF**>	Stokes U data objects
[STOKES_V]	<**NDF**>	Stokes V data objects

those with a principal data array, where only the components of its <**NDF**> structure are processed/copied to an output array, and (ii) those where the data arrays have equal importance, and the application, by convention, assumes the first <**NDF**> supplied contains the principal data array. There will be an HDS editor and <**NDF**> "dressing/undressing" utilities when this is not satisfactory. It is suggested that a common ADAM parameter name be assigned to this 'principal' <**NDF**>, *e.g.* MAIN_ARRAY.

11.1 Polarimetry Example

Stokes parameters are the most common method for storing and analysis of polarimetric data. Here is an illustrative example of how they might be stored using the <**NDF**> structure, taking the approach that the I data is the principal data array, and is therefore stored at the top-level of the structure. The Q, U and V-parameter data are <**NDF**> structures called, respectively, **[STOKES_Q]**, **[STOKES_U]** and **[STOKES_V]**, and located within a polarimetry extension.

The obvious alternative approach would be simply to add to the **[DATA_ARRAY]** an extra dimension so that the different Stokes parameters could all be stored in a single data array. Thus, for example, the four Stokes pictures from a 512×512 imaging polarimeter would be stored as different planes of a 4×512×512 data cube. Though superficially more elegant than using separate arrays for each Stokes parameter, such an approach would introduce the danger of invalid processing, because the Stokes parameters are intrinsically different from each other; they cannot be combined (for example adding a Q pixel value to its V value would be meaningless) whereas analogous arithmetic between values in the spatial time and wavelength/energy dimensions (for example rebinning) would, of course, be valid.

Usually, the different Stokes parameters will have the same axis information and, using the structure above, specialist polarimetry applications will be able to exploit this fact. However, general-purpose applications will not be able to do so, because of the rule on "tree-walking". To obtain other than default axis data using a general-purpose application, say displaying **[STOKES_Q.DATA_ARRAY]**, the axis information must be duplicated in the **[STOKES_Q]** structure; alternatively, application should have a parameter which specifies where the axis information is to be found. If the default is taken, the application should look for the <**AXIS**> structure in the normal place, *i.e.* within **[STOKES_Q.]**

11.2 Simplified <NDF> Structure

The support software associated with the standard data structures described above will take a long time to develop. In the meantime, some astronomers and programmers will want to convert their applications to ADAM and to start using HDS data structures. Therefore, a simple

Table 26: A polarimetry example using the <**NDF**>

Component Name	TYPE	Brief Description
[**TITLE**]	<**_CHAR**>	title of <**NDF**>
[**DATA_ARRAY**]	<**various**>	Stokes *I* data array
[**LABEL**]	<**_CHAR**>	label describing the data array
[**UNITS**]	<**_CHAR**>	units of the data array
[**VARIANCE**]	<**s_array**>	variance of the data array
[**QUALITY**]	<**various**>	quality of the data array
[**AXIS**(NAXIS)]	<**AXIS**>	axis values, labels, units and errors
		applicable to all Stokes parameters
[**HISTORY**]	<**HISTORY**>	history structure
[**MORE**]	<**EXT**>	extension structure
[**.POLARIMETRY**]	<**EXT**>	polarimetry extension

Table 27: Components of the Simple Extensible *n*-Dimensional-Data structure

Component Name	TYPE	Brief Description
[**TITLE**]	<**_CHAR**>	title of [**DATA_ARRAY**]
[**DATA_ARRAY**]	<**narray**>	NAXIS-dimensional data array
[**AXIS**(NAXIS)]	<**AXIS**>	axis values, labels, units and errors

and limited form of the <**NDF**> structure is available for their use. It will be comprehensible to the standard interfaces once they are ready, so that existing applications would then require minor modification, but the data files would not.

[TITLE] As described in the full <**NDF**>.

[DATA_ARRAY] This is the primary *n*-dimensional array of data values. It is the ONLY obligatory structure element. Note it can only be an array of numbers in the simplified <**NDF**>.

NAXIS No change from the standard <**NDF**>.

[AXIS] Note these are simplified <**AXIS**> structures. If they are not present, the pixel coordinates are used for the axis arrays, which have the same dimensions as the [**DATA_ARRAY**]

[DATA_ARRAY] Note this can only be an array of numbers. Mandatory.

[LABEL] No change from the normal <**AXIS**> structure.

Table 28: Components of the Simplified <**AXIS**> structure element

Component Name	TYPE	Brief Description
[**DATA_ARRAY**]	<**narray**>	axis value at each pixel
[**LABEL**]	<**_CHAR**>	axis label
[**UNITS**]	<**_CHAR**>	axis units

[UNITS] No change.

Applications must test pixels for magic values—there is no [**BAD_PIXEL**] flag in the simplified <**NDF**>.

12 Comparison with Wright-Giddings proposals

The term "structure" in the Wright-Giddings (WG) proposals meant a collection of related data objects, so that one level of the hierarchy could contain more than one "structure", *e.g.* GLOBAL and [**DATA_ARRAY**].

Of the GLOBAL structure only [**TITLE**] has been copied into the <**NDF**> structure.

WG components no longer used in <**NDF**> :

[DATA_MIN], [DATA_MAX] Following the precept of making life easier for applications programmers forces their exclusion. Principally they are used for scaling for display, yet they often fail to produce the desired result, especially for data with a wide dynamic range (as is often found in astronomy), because the extrema only give information about two pixels and are therefore unrepresentative. What often happens is that the display process is repeated several times (albeit more efficiently if [**DATA_MIN**] and [**DATA_MAX**] are stored), each time guessing the correct range more accurately. Even in those cases where the maximum and minimum provide a good scaling, their computation accounts for only about 20 per cent of the total time taken by the display process. A more-sophisticated approach at the outset pays off because the data need only be displayed once. The most successful methods are related to the distribution of pixel values. Histogram equalisation, central x per cent, or minus a to plus b standard deviations all work well.

Although there might be some hope that a min-max system could work if all algorithmic access to the datasets were through Starlink-supplied routines, it is inevitable that some programmers/users would forget to call the subroutines or fail to realise that their application may have affected the extrema.

[DATA_BLANK] This has been superseded by system-defined magic values, available to applications.

[DATA_LABEL], [DATA_UNITS] and [DATA_QUALITY] These have changed to allow standard subroutines to work on different labels, units and quality, *e.g.* in <**AXIS**>, and not be tied into a naming scheme, *e.g.* [**AXIS_LABEL**]. The interpretation of [**DATA_QUALITY**] has also changed for general-purpose applications.

Table 29: Comparison of <**NDF**> and the Wright-Giddings "DATA_ARRAY structure"

Wright-Giddings	NDF name	Comments
	[TITLE]	same name in WG GLOBAL
	[VARIANT]	no counterpart in WG
[DATA_ARRAY]	**[DATA_ARRAY]**	unchanged in name or meaning
[DATA_MIN]		no counterpart in <**NDF**>
[DATA_MAX]		no counterpart in <**NDF**>
[DATA_BLANK]		no counterpart in <**NDF**>
[DATA_SCALE]		**[SCALE]** in scaled <**narray**>
[DATA_ZERO]		**[ZERO]** in scaled <**narray**>
[DATA_LABEL]	**[LABEL]**	unchanged in meaning
[DATA_UNITS]	**[UNITS]**	unchanged in meaning
[DATA_QUALITY]	**[QUALITY]**	unchanged in meaning
	[BAD_PIXEL]	no counterpart in WG
[DATA_ERROR]	**[VARIANCE]**	variance rather than σ
[AXIS_CALIB]		no counterpart in <**NDF**>
[AXIS<n>_DATA]	**[AXIS(<n>).DATA_ARRAY]**	**[DATA_ARRAY]** for consistency
[AXIS<n>_UNITS]	**[AXIS(<n>).UNITS]**	unchanged in meaning
[AXIS<n>_LABEL]	**[AXIS(<n>).LABEL]**	unchanged in meaning
[AXIS<n>_ERROR]	**[AXIS(<n>).VARIANCE]**	variance rather than σ
[AXIS<n>_CALIB]		no counterpart in <**NDF**>
	[HISTORY]	no counterpart in WG
	[MORE]	no counterpart though it can store WG GLOBAL objects

[DATA_ERROR] This has changed as for **[DATA_LABEL]** *etc.*, but also because the interpretation has changed to variance.

[AXIS_CALIB] and [AXIS<n>_CALIB] These are too specialised and the processing rules are unknown for general-purpose applications. Calibration data should be situated in **[MORE]** within an extension, perhaps which shadows the main **[AXIS]** structures.

13 Creating new structures

The methods presented in this section are intended to ensure that the rules given in Section 2 are adhered to. Particularly, it makes sure that standard structures are used wherever possible, and therefore encourages the building of new structures (when needed) by gathering together existing structures, so ensuring the maximum commonality.

New structures must not be constructed simply by adding new components into standard ones (which is illegal), but instead by adding a new layer. The HDS hierarchy then provides a natural barrier between separate structures, and ensures that further components can later be added at any level without risking naming conflicts.

13.1 Definitions

In the description of the design process the total hierarchy of structures is called a *dataset* to distinguish it from a single component structure. It is equivalent to the contents of an HDS container file. The term *structure* refers to a set of related data items; it corresponds to a single level of a dataset. Often a dataset will consist simply of a single structure.

13.2 Algorithm

A summary of the algorithm to be used when creating a new dataset is given below. The circled numbering refers to expanded notes in the next subsection.

Define what the software is going to do and not going to do ①
Identify, in concept, the datasets required
Determine their interrelations, perhaps via a tree diagram ②
Start at the most deeply nested level of the hierarchy
for each structure
  Identify the data components ③
  if an existing standard structure can be used **then**
    Use it
    Place remaining associated items in a new structure or in an extension
  else
    Assign a unique HDS TYPE to the new structure ④
    Assign a NAME to each component ⑤
    Determine the rules and restrictions governing the way the data will be stored in the various components

```
        Assign a TYPE to each component      ⑥
        Identify the sorts of operation to be performed on the structure and ensure they are
          meaningfully defined      ⑦
        if the processing of a component cannot be defined in some cases then remove the
          component from the structure
        Implement and document the software needed to process the new structure      ⑧
        if the new structure is to become a standard type then
            Submit it and its software to Starlink for approval      ⑨
        endif
    endif
endfor
```

13.3 Explanatory Notes

(1)
- It is important to define the scope of the software initially and not to let it expand arbitrarily during implementation. If the software design does subsequently need to be revised, then the dataset may also need re-designing.
- During the following stages of the design process, the original outline of the dataset may prove to be incorrect or inadequate, especially in more-complex hierarchies. In such cases you have to start again.

(2)
- The interrelations between structures specify how they should be organised hierarchically. Drawing a dendrogram should help.
- The design process is bottom-up. Multiple-level datasets are built up from from the lowest (most deeply nested) level of the hierarchy. Design each and every structure at the current HDS level before going to the next higher level.

(3)
- Check to see whether any of the required data components are standard structures or are components of standard structures. If suitable standard structures already exist, then use them. If not, and you have to design new structures, try to make them general so that they might later become standard structures themselves.
- The original dendrogram design of the dataset may grow some extra branches if standard structures can be used, because there may be a net increase in the number of structures.
- Certain standard structures include provision for extension structures, and may thus be used even if there is no appropriate place in the standard structure itself for some of the items to be stored.
- Using existing standard structures gives the obvious advantages of being able to use existing software. (Starlink will maintain a list of standard structures and their conventions.) The standards and conventions associated with standard structures must be observed by all new software which uses them.

(4)
- The TYPE should reflect the sort of data to be held in the structure, but must not conflict with the TYPE of any other standard data structure. Starlink management should be consulted when defining TYPEs.

(5)
- The names should preferably identify the rôle which each component plays. Although the name will have no global significance outside the structure, it may still be sensible to have a naming convention for certain common types of structure to avoid confusion.
- There may be any number of rules and conventions governing use of the structure. For instance, some components may be optional, and the presence of some components may depend on others (as with the VARIANT concept). These rules must be explicitly stated and obeyed by all software which uses the structure. If this software is likely to be written by many different people, then the rules should obviously be kept simple.

(6)
- Only primitives or structures of a TYPE already defined may be used.
- Since only defined TYPEs (which includes primitives) may be used, any substructures must already have been defined along with the rules for processing them. It might also occasionally be appropriate to define structures "recursively" by including components of the same TYPE as that being defined.
- Often, standard subroutines will already exist for processing the data components from which the new structure is being built, and these can therefore be used to process the components of the new structure.

(7)
- Ensure that all the operations are meaningfully defined in terms of what will happen to each component when the structure is processed. Consider all valid combinations of structure components.
- Many packages "grow" indefinitely, so it may not be possible to enumerate all possible operations. However, if the initial (global) stage of the design was obeyed, it should be possible to identify them as broad classes, such as [image display, arithmetic, spatial smoothing...], or [create history, append history, search for history record...].
- It may be necessary to reject some components if you cannot meaningfully define what will happen to them in all circumstances.

(8)
- The software should obey all the conventions appropriate to the new structure (and any other structures it uses). When accessing a structure, software should first check its TYPE—this specifies how the structure contents are to be interpreted. Any component not covered by the structure definition should be completely ignored.
- From time to time, ignorance and independence of spirit will no doubt lead implementors and users into inserting extra components into a structure, but these are illegal and will be ignored. This is **not** a valid way of defining a new structure.

(9)
- If the new structure is to become a standard type, submit the design (providing details of the NAME, TYPE, meaning and processing rules for each data object) to the Starlink Head of Applications for approval and registration. If appropriate, a subroutine interface should be written for handling the structure; this would ensure that the conventions governing its use are enforced. Any associated software should also be submitted to the Head of Applications.
- Once a new standard structure has been accepted, anyone is free to use the structure and to incorporate it in any new structures he or she may create. Once this point is reached, it may be difficult to change the structure definition without upsetting somebody; changes in the form of additions to the structure are the least likely to cause trouble.

13.4 Extensions

Once a "core" of fairly simple standard structures exists, the process of designing more specialised structures will be devolved to the various SIGs, who can use the simpler structures as building blocks. This avoids the problems of the 'all or nothing' monolithic approach. When a more complex (and therefore highly specialised) structure is built out of simpler ones, software will then automatically exist for processing all its substructures in a more general way. This should give a high degree of flexibility.

There will be independent extensions, each having a uniquely defined TYPE together with rules for its interpretation. Though many extensions will be independent and self-contained, some will form hierarchies. The design of each extension should be kept straightforward and appropriate to the kind of software which will use it. Simple and specialised, simple and general, and complex and specialised are all acceptable, but implementors should beware of attempting to design extensions which are both complex and general. By introducing a strict criterion to decide whether a given component is acceptable ("do we know how to process it?"), it is ensured that the problem is broken into manageable pieces, the complexity of which does not exceed our software-writing abilities.

14 Acknowledgments

Parts of this document have been adapted from earlier proposals. The contributions of Keith Shortridge and Trevor Ponman were particularly helpful. The comments, criticisms and ideas of the participants of the Great Debate greatly improved the content of this document.

15 References

Wright, S.L. & Giddings, J.R. 1983 *Standard Starlink Data Structures* (draft proposal).

Wells, D.C. & Greisen, E.W. 1979 in *Image Processing in Astronomy*, eds. Sedmak, K., Capaccioli, M. & Allen, R.J., Osservatorio Astronomico di Trieste, p.445.

A Release Notes

The document is substantially unchanged from the 1989 October version. The recent changes (SGP/38.3) are:

- a correction to the formula for evalution of the normalised co-ordinates for a Chebyshev polynomial,
- the addition of a <**POLYNOMIAL**> variant for B-Splines, and
- some formatting changes.

Note that this document does not describe additional structures created since the late 1990s that are commonly found in NDFs, such as WCS and PROVENANCE.