Draft date: 14 March 2005

8.3.5 Variable-Length Arrays

One of the most attractive features of binary tables is that any field of the table can be an array. In the standard case this is a fixed size array, i.e., a fixed amount of storage is allocated in each record for the array data--whether it is used or not. This is fine so long as the arrays are small or a fixed amount of array data will be stored in each record, but if the stored array length varies for different records, it is necessary to impose a fixed upper limit on the size of the array that can be stored. If this upper limit is made too large excessive wasted space can result and the binary table mechanism becomes seriously inefficient. If the limit is set too low then it may become impossible to store certain types of data in the table.

The ``variable-length array'' construct presented here was devised to deal with this problem. Variable-length arrays are implemented in such a way that, even if a table contains such arrays, a simple reader program which does not understand variable-length arrays will still be able to read the main table (in other words a table containing variable-length arrays conforms to the basic binary table standard). The implementation chosen is such that the records in the main table remain fixed in size even if the table contains a variable-length array field, allowing efficient random access to the main table.

Variable-length arrays are logically equivalent to regular static arrays, the only differences being 1) the length of the stored array can differ for different records, and 2) the array data are not stored directly in the table records. Since a field of any datatype can be a static array, a field of any datatype can also be a variable-length array (excluding type `P`, the variable-length array descriptor itself, which is not a datatype so much as a storage class specifier). Other established FITS conventions that apply to static arrays will generally apply as well to variable-length arrays.

A variable-length array is declared in the table header with a special field datatype specifier of the form

$$\texttt{r}\texttt{P}\texttt{t}\texttt{(}e_{max}\texttt{)}$$

where the ``P'' indicates the amount of space occupied by the array descriptor in the data record (64 bits), the element count `r` should be 0, 1, or absent, `t` is a character denoting the datatype of the array data (`L`, `X`, `B`, `I`, `J`, etc., but not `P`), and e_{max} is a quantity guaranteed to be equal to or greater than the maximum number of elements of type `t` actually stored in a table record. There is no built-in upper limit on the size of a stored array (other than the fundamental limit imposed by the range of the 32-bit array descriptor, defined below); e_{max} merely reflects the size of the largest array actually stored in the table, and is provided to avoid the need to preview the table when, for example, reading a table containing variable-length elements into a database that supports only fixed size arrays. There may be additional characters in the `TFORMn` keyword following the e_{max}.

For example,

```
TFORM8 = 'PB(1800)' / Variable byte array
```

indicates that field 8 of the table is a variable-length array of type byte, with a maximum stored array length not to exceed 1800 array elements (bytes in this case).

The data for the variable-length arrays in a table are not stored in the actual data records; they are stored in a special data area, the heap, following the last fixed size data record. What is stored in the data record is an *array descriptor*. This consists of two 32-bit signed integer values: the number of elements (array length) of the stored array, followed by the zero-indexed byte offset of the first element of the array, measured from the start of the heap area. The meaning of a negative value for either of these integers is not defined by this standard. Storage for the array is contiguous. The array descriptor for field *N* as it would appear embedded in a data record is illustrated symbolically below:

```
...[field N-1][(nelem,offset)][field N+1]...
```

If the stored array length is zero there is no array data, and the offset value is undefined (it should be set to zero). The storage referenced by an array descriptor must lie entirely within the heap area; negative offsets are not permitted.

A binary table containing variable-length arrays consists of three principal segments, as follows:

[table_header] [record_storage_area] [heap_area]

The table header consists of one or more 2880-byte *FITS* logical records with the last record indicated by the keyword `END` somewhere in the record. The record storage area begins with the next 2880-byte logical record following the last header record and is `NAXIS1 × NAXIS2` bytes in length. The zero indexed byte offset of the heap measured from the start of the record storage area is given by the `THEAP` keyword in the header. If this keyword is missing the heap is assumed to begin with the byte immediately following the last data record, otherwise there may be a gap between the last stored record and the start of the heap. If there is no gap the value of the heap offset is `NAXIS1 × NAXIS2`. The total length in bytes of the heap area following the last stored record (gap plus heap) is given by the `PCOUNT` keyword in the table header.

For example, suppose we have a table containing 5 rows each 168 bytes long , with a heap area 3000 bytes long, beginning at an offset of 2880, thereby aligning the record storage and heap areas on *FITS* record boundaries (this alignment is not necessarily recommended but is useful for our example). The data portion of the table consists of 3 2880-byte *FITS* records: the first record contains the 840 bytes from the 5 rows of the main table followed by 2040 fill bytes; the heap completely fills the second record; the third record contains the remaining 120 bytes of the heap followed by 2760 fill bytes. PCOUNT gives the total number of bytes from the end of the main table to the end of the heap and in this example has a value of 2040 + 2880 + 120 = 5040. This is expressed in the table header as:

```
NAXIS1  =    168 / Width of table row in bytes
NAXIS2  =      5 / Number of rows in table
PCOUNT  =   5040 / Random parameter count
  ...
THEAP   =   2880 / Byte offset of heap area
```

The values of `TSCALn` and `TZEROn` for variable-length array column entries are to be applied to the values in the data array in the heap area, not the values of the array

descriptor. These keywords can be used to scale data values in either static or variable-length arrays.

While the above description is sufficient to define the required features of the variable-length array implementation, some hints regarding usage of the variable-length array facility may also be useful.

Programs which read binary tables should take care to not assume more about the physical layout of the table than is required by the specification. For example, there are no requirements on the alignment of data within the heap. If efficient runtime access is a concern one may want to design the table so that data arrays are aligned to the size of an array element. In another case one might want to minimize storage and forgo any efforts at alignment (by careful design it is often possible to achieve both goals). Variable-length array data may be stored in the heap in any order, i.e., the data for record N+1 are not necessarily stored at a larger offset than that for record N. There may be gaps in the heap where no data are stored. Pointer aliasing is permitted, i.e., the array descriptors for two or more arrays may point to the same storage location (this could be used to save storage if two or more arrays are identical).

Byte arrays are a special case because they can be used to store a ``typeless'' data sequence. Since *FITS* is a machine-independent storage format, some form of machine-specific data conversion (byte swapping, floating point format conversion) is implied when accessing stored data with types such as integer and floating, but byte arrays are copied to and from external storage without any form of conversion.

An important feature of variable-length arrays is that it is possible that the stored array length may be zero. This makes it possible to have a column of the table for which, typically, no data are present in each stored record. When data are present the stored array can be as large as necessary. This can be useful when storing complex objects as records in a table.

Accessing a binary table stored on a random access storage medium is straightforward. Since the data records in the main table are fixed in size they may be randomly accessed given the record number, by computing the offset. Once the record has been read in, any variable-length array data may be directly accessed using the element count and offset given by the array descriptor stored in the data record.

Reading a binary table stored on a sequential access storage medium requires that a table of array descriptors be built up as the main table records are read in. Once all the table records have been read, the array descriptors are sorted by the offset of the array data in the heap. As the heap data are read, arrays are extracted sequentially from the heap and stored in the affected records using the back pointers to the record and field from the table of array descriptors. Since array aliasing is permitted, it may be necessary to store a given array in more than one field or record.

Variable-length arrays are more complicated than regular static arrays and may not be supported by some software systems. The producers of FITS data products should consider the capabilities of the likely recipients of their files when deciding whether or not to use this format, and as a general rule should use it only in cases where it provides significant advantages over the simpler fixed length array format. In particular, the use of variable-length arrays may present difficulties for applications that ingest the FITS file via a sequential input stream because the application cannot fully process any rows in the

table until after the entire fixed length table and potentially the entire heap has been transmitted as outlined in the previous paragraph.