for that zVariable. After the read the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). An error is returned if the current sequential value is past the last record for the zVariable. Required arguments are as follows:

out: `void *value`

Value. This buffer must be large enough to hold the value. The value is read from the CDF and placed into memory at address `value`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are read.

<GET_,zVARs_MAXREC_>

Inquires the maximum record number of the zVariables in the current CDF. Note that this is not the number of records but rather the maximum record number (which is one less than the number of records). A value of negative one (-1) indicates that the zVariables contain no records. The maximum record number for an individual zVariable may be inquired using <GET_,zVAR_MAXREC_>. Required arguments are as follows:

out: `long *maxRec`

Maximum record number.

The only required preselected object/state is the current CDF.

<GET_,zVARs_RECDATA_>

Reads full-physical records from one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is read at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of zVariables from which to read. This must be at least one (1).

in: `long varNums[]`

The zVariables from which to read. This array, whose size is determined by the value of `numVars`, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: `void *buffer`

The buffer into which the full-physical zVariable records being read are to be placed. This buffer must be large enough to hold the full-physical records. The order of the full-physical zVariable records in this buffer will correspond to the zVariable numbers listed in `varNums`, and this buffer will be contiguous — there will be no spacing between full-physical zVariable records. Be careful if using C `struct` objects to receive multiple full-physical zVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to allocate this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`<OPEN_,CDF_>`

Opens the named CDF. The opened CDF implicitly becomes the current CDF. Required arguments are as follows:

in: `char *CDFpath`

Pathname of the CDF to be opened. (Do not append an extension.) This can be at most `CDF_PATHNAME_LEN` characters (excluding the `NUL` terminator). A CDF pathname may contain disk and directory specifications that conform to the conventions of the operating system being used (including logical names on VMS systems and environment variables on UNIX systems).

UNIX: Pathnames are case-sensitive.

out: `CDFid *id`

CDF identifier to be used in subsequent operations on the CDF.

There are no required preselected objects/states.

`<PUT_,ATTR_NAME_>`

Renames the current attribute (in the current CDF). An attribute with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *attrName`

New attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

The required preselected objects/states are the current CDF and its current attribute.

`<PUT_,ATTR_SCOPE_>`

Respecifies the scope for the current attribute (in the current CDF). Required arguments are as follows:

in: `long scope`

New attribute scope. Specify one of the scopes described in Section 4.10.

The required preselected objects/states are the current CDF and its current attribute.

`<PUT_,CDF_ENCODING_>`

Respecifies the data encoding of the current CDF. A CDF's data encoding may not be changed after any variable values (including the pad value) or attribute entries have been written. Required arguments are as follows:

in: `long encoding`

New data encoding. Specify one of the encodings described in Section 4.6.

The only required preselected object/state is the current CDF.

<PUT_,CDF_FORMAT_>

Respecifies the format of the current CDF. A CDF's format may not be changed after any variables have been created. Required arguments are as follows:

in: `long format`

New CDF format. Specify one of the formats described in Section 4.4.

The only required preselected object/state is the current CDF.

<PUT_,CDF_MAJORITY_>

Respecifies the variable majority of the current CDF. A CDF's variable majority may not be changed after any variable values have been written. Required arguments are as follows:

in: `long majority`

New variable majority. Specify one of the majorities described in Section 4.8.

The only required preselected object/state is the current CDF.

<PUT_,gENTRY_DATA_>

Writes a gEntry to the current attribute at the current gEntry number (in the current CDF). An existing gEntry may be overwritten with a new gEntry having the same data specification (data type and number of elements) or a different data specification. Required arguments are as follows:

in: `long dataType`

Data type of the gEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character string data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: `void *value`

Value(s). The entry value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

<PUT_,gENTRY_DATASPEC_>

Modifies the data specification (data type and number of elements) of the gEntry at the current gEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

in: `long dataType`

New data type of the gEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current gEntry number.

NOTE: Only use this operation on gAttributes. An error will occur if used on a vAttribute.

`<PUT_,rENTRY_DATA_>`

Writes an rEntry to the current attribute at the current rEntry number (in the current CDF). An existing rEntry may be overwritten with a new rEntry having the same data specification (data type and number of elements) or a different data specification. Required arguments are as follows:

in: `long dataType`

Data type of the rEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character string data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: `void *value`

Value(s). The entry value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,rENTRY_DATASPEC_>`

Modifies the data specification (data type and number of elements) of the rEntry at the current rEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

in: `long dataType`

New data type of the rEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current rEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,rVAR_ALLOCATERECS_>`

Specifies the number of records to allocate for the current rVariable (in the current CDF). This operation is only applicable to single-file CDFs. This may be specified only once per rVariable and before any records have been written to that rVariable. The Concepts chapter in the CDF User's Guide describes the allocation of variable records Required arguments are as follows:

in: `long NallocateRecs`

Number of records to allocate.

The required preselected objects/states are the current CDF and its current rVariable.

`<PUT_,rVAR_DATA_>`

Writes one value to the current rVariable (in the current CDF). The value is written at the current record number and current dimension indices for the rVariables (in the current CDF). Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current rVariable, its current record number for rVariables, and its current dimension indices for rVariables.

`<PUT_,rVAR_DATASPEC_>`

Respecifies the data specification (data type and number of elements) of the current rVariable (in the current CDF). An rVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

in: `long dataType`

New data type. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type at each value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in each string (an array of characters). A string exists at each value. For the non-character string data types this must be one (1) — arrays of values are not allowed for non-character string data types.

The required preselected objects/states are the current CDF and its current rVariable.

`<PUT_,rVAR_DIMVARYS_>`

Respecifies the dimension variances of the current rVariable (in the current CDF). An rVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long dimVarys[]`

New dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_,rVAR_EXTENDRECS_>

Specifies the number of physical records to extend the current rVariable (in the current CDF) whenever that rVariable needs to be extended. The Concepts chapter in the CDF User's Guide describes extend records. **NOTE:** The number of extend records has no effect for NRV rVariables or multi-file CDFs. Required arguments are as follows:

in: `long NextendRecs`

Number of records to extend. A value of zero (0) indicates that the default number of records should be used.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_,rVAR_HYPERDATA_>

Writes one or more values to the current rVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for the rVariables (in the current CDF). Required arguments are as follows:

in: `void *buffer`

Values. The values starting at memory address `buffer` are written to the CDF.

The required preselected objects/states are the current CDF, its current rVariable, its current record number, record count, and record interval for rVariables, and its current dimension indices, dimension counts, and dimension intervals for rVariables.

<PUT_,rVAR_INITIALRECS_>

Specifies the number of records to initially write to the current rVariable (in the current CDF). This may be specified only once per rVariable and before any other records have been written to that rVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: `long NinitialRecs`

Number of records to write.

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_,rVAR_NAME_>

Renames the current rVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *varName`

New name of the rVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The required preselected objects/states are the current CDF and its current rVariable.

<PUT_,rVAR_PADVALUE_>

Specifies the pad value for the current rVariable (in the current CDF). An rVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). The Concepts chapter in the CDF User's Guide describes variable pad values. Required arguments are as follows:

in: `void *value`

Pad value. The pad value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF and its current rVariable.

`<PUT_,rVAR_RECVARY_>`

Respecifies the record variance of the current rVariable (in the current CDF). An rVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `long recVary`

New record variance. Specify one of the variances described in Section 4.9.

The required preselected objects/states are the current CDF and its current rVariable.

`<PUT_,rVAR_SEQDATA_>`

Writes one value to the current rVariable (in the current CDF) at the current sequential value for that rVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the rVariable, the rVariable is extended as necessary. Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current rVariable, and the current sequential value for the rVariable. Note that the current sequential value for an rVariable increments automatically as values are written.

`<PUT_,rVARs_RECDATA_>`

Writes full-physical records to one or more rVariables (in the current CDF). The full-physical records are written at the current record number for rVariables. This operation does not affect the current rVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of rVariables to which to write. This must be at least one (1).

in: `long varNums[]`

The rVariables to which to write. This array, whose size is determined by the value of `numVars`, contains rVariable numbers. The rVariable numbers can be listed in any order.

in: `void *buffer`

The buffer of full-physical rVariable records to be written. The order of the full-physical rVariable records in this buffer must agree with the rVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical rVariable records. Be careful if using C `struct` objects to store multiple full-physical rVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `sturct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to create this buffer.

The required preselected objects/states are the current CDF and its current record number for rVariables.

`<PUT_,zENTRY_DATA_>`

Writes a zEntry to the current attribute at the current zEntry number (in the current CDF). An existing zEntry may be overwritten with a new zEntry having the same data specification (data type and number of elements) or a different data specification. Required arguments are as follows:

in: `long dataType`

Data type of the zEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type. This may be greater than one (1) for any of the supported data types. For character string data types (`CDF_CHAR` and `CDF_UCHAR`) this is the number of characters in the string (an array of characters). For all other data types this is the number of elements in an array of that data type.

in: `void *value`

Value(s). The entry value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,zENTRY_DATASPEC_>`

Modifies the data specification (data type and number of elements) of the zEntry at the current zEntry number of the current attribute (in the current CDF). The new and old data types must be equivalent, and the number of elements must not be changed. Equivalent data types are described in the Concepts chapter in the CDF User's Guide. Required arguments are as follows:

in: `long dataType`

New data type of the zEntry. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type.

The required preselected objects/states are the current CDF, its current attribute, and its current zEntry number.

NOTE: Only use this operation on vAttributes. An error will occur if used on a gAttribute.

`<PUT_,zVAR_ALLOCATERECS_>`

Specifies the number of records to allocate for the current zVariable (in the current CDF). This operation is only applicable to single-file CDFs. This may be specified only once per zVariable and before any records have been written to that zVariable. The Concepts chapter in the CDF User's Guide describes the allocation of variable records Required arguments are as follows:

in: `long NallocateRecs`

Number of records to allocate.

The required preselected objects/states are the current CDF and its current zVariable.

`<PUT_,zVAR_DATA_>`

Writes one value to the current zVariable (in the current CDF). The value is written at the current record number and current dimension indices for that zVariable (in the current CDF). Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current zVariable, the current record number for the zVariable, and the current dimension indices for the zVariable.

`<PUT_,zVAR_DATASPEC_>`

Respecifies the data specification (data type and number of elements) of the current zVariable (in the current CDF). A zVariable's data specification may not be changed if the new data specification is not equivalent to the old data specification and any values (including the pad value) have been written. Data specifications are considered equivalent if the data types are equivalent (see the Concepts chapter in the CDF User's Guide) and the number of elements are the same. Required arguments are as follows:

in: `long dataType`

New data type. Specify one of the data types described in Section 4.5.

in: `long numElements`

Number of elements of the data type at each value. For character string data types (`CDF_CHAR` and `CDF_UCHAR`), this is the number of characters in each string (an array of characters). A string exists at each value. For the non-character string data types this must be one (1) — arrays of values are not allowed for non-character string data types.

The required preselected objects/states are the current CDF and its current zVariable.

`<PUT_,zVAR_DIMVARYS_>`

Respecifies the dimension variances of the current zVariable (in the current CDF). A zVariable's dimension variances may not be changed if any values have been written (except for an explicit pad value — it may have been written). For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long dimVarys[]`

New dimension variances. `dimVarys` is a 1-dimensional array containing one element per dimension. Each element of `dimVarys` specifies the corresponding dimension variance. For each dimension specify one of the variances described in Section 4.9.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_EXTENDRECS_>

Specifies the number of physical records to extend the current zVariable (in the current CDF) whenever that zVariable needs to be extended. The Concepts chapter in the CDF User's Guide describes extend records. **NOTE:** The number of extend records has no effect for NRV zVariables or multi-file CDFs. Required arguments are as follows:

in: `long NextendRecs`

Number of records to extend. A value of zero (0) indicates that the default number of records should be used.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_INITIALRECS_>

Specifies the number of records to initially write to the current zVariable (in the current CDF). This may be specified only once per zVariable and before any other records have been written to that zVariable. If a pad value has not yet been specified, the default is used (see the Concepts chapter in the CDF User's Guide). If a pad value has been explicitly specified, that value is written to the records. The Concepts chapter in the CDF User's Guide describes initial records. Required arguments are as follows:

in: `long NinitialRecs`

Number of records to write.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_HYPERDATA_>

Writes one or more values to the current zVariable (in the current CDF). The values are written based on the current record number, current record count, current record interval, current dimension indices, current dimension counts, and current dimension intervals for that zVariable (in the current CDF). Required arguments are as follows:

in: `void *buffer`

Values. The values starting at memory address `buffer` are written to the CDF.

The required preselected objects/states are the current CDF, its current zVariable, the current record number, record count, and record interval for the zVariable, and the current dimension indices, dimension counts, and dimension intervals for the zVariable.

<PUT_,zVAR_NAME_>

Renames the current zVariable (in the current CDF). A variable (rVariable or zVariable) with the same name must not already exist in the CDF. Required arguments are as follows:

in: `char *varName`

New name of the zVariable. This may consist of at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_PADVALUE_>

Specifies the pad value for the current zVariable (in the current CDF). A zVariable's pad value may be specified (or respecified) at any time without affecting already written values (including where pad values were used). The Concepts chapter in the CDF User's Guide describes variable pad values. Required arguments are as follows:

in: `void *value`

Pad value. The pad value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_RECVARY_>

Respecifies the record variance of the current zVariable (in the current CDF). A zVariable's record variance may not be changed if any values have been written (except for an explicit pad value — it may have been written). Required arguments are as follows:

in: `long recVary`

New record variance. Specify one of the variances described in Section 4.9.

The required preselected objects/states are the current CDF and its current zVariable.

<PUT_,zVAR_SEQDATA_>

Writes one value to the current zVariable (in the current CDF) at the current sequential value for that zVariable. After the write the current sequential value is automatically incremented to the next value (crossing a record boundary if necessary). If the current sequential value is past the last record for the zVariable, the zVariable is extended as necessary. Required arguments are as follows:

in: `void *value`

Value. The value is written to the CDF from memory address `value`.

The required preselected objects/states are the current CDF, its current zVariable, and the current sequential value for the zVariable. Note that the current sequential value for a zVariable increments automatically as values are written.

<PUT_,zVARs_RECDATA_>

Writes full-physical records to one or more zVariables (in the current CDF). The full-physical record for a particular zVariable is written at the current record number for that zVariable. (The record numbers do not have to be the same but in most cases probably will be.) This operation does not affect the current zVariable (in the current CDF). Required arguments are as follows:

in: `long numVars`

The number of zVariables to which to write. This must be at least one (1).

in: `long varNums[]`

The zVariables to which to write. This array, whose size is determined by the value of `numVars`, contains zVariable numbers. The zVariable numbers can be listed in any order.

in: `void *buffer`

The buffer of full-physical zVariable records to be written. The order of the full-physical zVariable records in this buffer must agree with the zVariable numbers listed in `varNums`, and this buffer must be contiguous — there can be no spacing between full-physical zVariable records. Be careful if using C `struct` objects to store multiple full-physical zVariable records. C compilers on some operating systems will pad between the elements of a `struct` in order to prevent memory alignment errors (i.e., the elements of a `struct` may not be contiguous). See the Concepts chapter in the CDF User's Guide for more details on how to create this buffer.

The required preselected objects/states are the current CDF and the current record number for each of the zVariables specified. A convenience operation exists, `<<SELECT_,zVARs_RECNUMBER_>`, that allows the current record number for each zVariable to be selected at one time (as opposed to selecting the current record numbers one at a time using `<SELECT_,zVAR_RECNUMBER_>`).

`<SELECT_,ATTR_>`

Explicitly selects the current attribute (in the current CDF) by number. Required arguments are as follows:

in: `long attrNum`

Attribute number.

The only required preselected object/state is the current CDF.

`<SELECT_,ATTR_NAME_>`

Explicitly selects the current attribute (in the current CDF) by name. Required arguments are as follows:

in: `char *attrName`

Attribute name. This may be at most `CDF_ATTR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`<SELECT_,CDF_>`

Explicitly selects the current CDF. Required arguments are as follows:

in: `CDFid id`

Identifier of the CDF. This identifier must have been initialized by a successful `<CREATE_,CDF_>` or `<OPEN_,CDF_>` operation.

There are no required preselected objects/states.

`<SELECT_,CDF_CACHESIZE_>`

Selects the number of cache buffers to be used for the "`.cdf`" file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`<SELECT_,CDF_DECODING_>`

Selects a decoding (for the current CDF). Required arguments are as follows:

in: `long decoding`

The decoding. Specify one of the decodings described in Section 4.7.

The only required preselected object/state is the current CDF.

`<SELECT_,CDF_NEGtoPOSfp0_MODE_>`

Selects a `-0.0` to `0.0` mode (for the current CDF). Required arguments are as follows:

in: `long mode`

The `-0.0` to `0.0` mode. Specify one of the `-0.0` to `0.0` modes described in Section 4.13.

The only required preselected object/state is the current CDF.

`<SELECT_,CDF_READONLY_MODE_>`

Selects a read-only mode (for the current CDF). Required arguments are as follows:

in: `long mode`

The read-only mode. Specify one of the read-only modes described in Section 4.11.

The only required preselected object/state is the current CDF.

`<SELECT_,CDF_STATUS_>`

Selects the current status code. Required arguments are as follows:

in: `CDFstatus status`

CDF status code.

There are no required preselected objects/states.

`<SELECT_,CDF_zMODE_>`

Selects a zMode (for the current CDF). Required arguments are as follows:

in: `long mode`

The zMode. Specify one of the zModes described in Section 4.12.

The only required preselected object/state is the current CDF.

`<SELECT_,gENTRY_>`

Selects the current gEntry number for all gAttributes in the current CDF. Required arguments are as follows:

in: `long entryNum`

gEntry number.

The only required preselected object/state is the current CDF.

<SELECT_,rENTRY_>

Selects the current rEntry number for all vAttributes in the current CDF. Required arguments are as follows:

in: `long entryNum`

rEntry number.

The only required preselected object/state is the current CDF.

<SELECT_,rENTRY_NAME_>

Selects the current rEntry number for all vAttributes (in the current CDF) by rVariable name. The number of the named rVariable becomes the current rEntry number. (The current rVariable is not changed.) Required arguments are as follows:

in: `char *varName`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

<SELECT_,rVAR_>

Explicitly selects the current rVariable (in the current CDF) by number. Required arguments are as follows:

in: `long varNum`

rVariable number.

The only required preselected object/state is the current CDF.

<SELECT_,rVAR_CACHESIZE_>

Selects the number of cache buffers to be used for the current rVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The required preselected objects/states are the current CDF and its current rVariable.

<SELECT_,rVAR_NAME_>

Explicitly selects the current rVariable (in the current CDF) by name. Required arguments are as follows:

in: `char *varName`

rVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`<SELECT_,rVAR_SEQPOS_>`

Selects the current sequential value for sequential access for the current rVariable (in the current CDF). Note that a current sequential value is maintained for each rVariable individually. Required arguments are as follows:

in: `long recNum`

Record number.

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per rVariable dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current rVariable.

`<SELECT_,rVARs_CACHESIZE_>`

Selects the number of cache buffers to be used for all of the rVariable files (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

`<SELECT_,rVARs_DIMCOUNTS_>`

Selects the current dimension counts for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long counts[]`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` specifies the corresponding dimension count.

The only required preselected object/state is the current CDF.

`<SELECT_,rVARs_DIMINDICES_>`

Selects the current dimension indices for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index.

The only required preselected object/state is the current CDF.

`<SELECT_,rVARs_DIMINTERVALS_>`

Selects the current dimension intervals for all rVariables in the current CDF. For 0-dimensional rVariables this operation is not applicable. Required arguments are as follows:

in: `long intervals[]`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` specifies the corresponding dimension interval.

The only required preselected object/state is the current CDF.

`<SELECT_,rVARs_RECCOUNT_>`

Selects the current record count for all rVariables in the current CDF. Required arguments are as follows:

in: `long recCount`

Record count.

The only required preselected object/state is the current CDF.

`<SELECT_,rVARs_RECINTERVAL_>`

Selects the current record interval for all rVariables in the current CDF. Required arguments are as follows:

in: `long recInterval`

Record interval.

The only required preselected object/state is the current CDF.

`<SELECT_,rVARs_RECNUMBER_>`

Selects the current record number for all rVariables in the current CDF. Required arguments are as follows:

in: `long recNum`

Record number.

The only required preselected object/state is the current CDF.

`<SELECT_,zENTRY_>`

Selects the current zEntry number for all vAttributes in the current CDF. Required arguments are as follows:

in: `long entryNum`

zEntry number.

The only required preselected object/state is the current CDF.

`<SELECT_,zENTRY_NAME_>`

Selects the current zEntry number for all vAttributes (in the current CDF) by zVariable name. The number of the named zVariable becomes the current zEntry number. (The current zVariable is not changed.) Required arguments are as follows:

in: `char *varName`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

`<SELECT_,zVAR_>`

Explicitly selects the current zVariable (in the current CDF) by number. Required arguments are as follows:

in: `long varNum`

zVariable number.

The only required preselected object/state is the current CDF.

`<SELECT_,zVAR_CACHESIZE_>`

Selects the number of cache buffers to be used for the current zVariable's file (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The required preselected objects/states are the current CDF and its current zVariable.

`<SELECT_,zVAR_DIMCOUNTS_>`

Selects the current dimension counts for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long counts[]`

Dimension counts. `counts` is a 1-dimensional array containing one element per dimension. Each element of `counts` specifies the corresponding dimension count.

The required preselected objects/states are the current CDF and its current zVariable.

`<SELECT_,zVAR_DIMINDICES_>`

Selects the current dimension indices for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index.

The required preselected objects/states are the current CDF and its current zVariable.

`<SELECT_,zVAR_DIMINTERVALS_>`

Selects the current dimension intervals for the current zVariable in the current CDF. For 0-dimensional zVariables this operation is not applicable. Required arguments are as follows:

in: `long intervals[]`

Dimension intervals. `intervals` is a 1-dimensional array containing one element per dimension. Each element of `intervals` specifies the corresponding dimension interval.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_,zVAR_NAME_>

Explicitly selects the current zVariable (in the current CDF) by name. Required arguments are as follows:

in: `char *varName`

zVariable name. This may be at most `CDF_VAR_NAME_LEN` characters (excluding the `NUL` terminator).

The only required preselected object/state is the current CDF.

<SELECT_,zVAR_RECCOUNT_>

Selects the current record count for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recCount`

Record count.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_,zVAR_RECINTERVAL_>

Selects the current record interval for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recInterval`

Record interval.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_,zVAR_RECNUMBER_>

Selects the current record number for the current zVariable in the current CDF. Required arguments are as follows:

in: `long recNum`

Record number.

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_,zVAR_SEQPOS_>

Selects the current sequential value for sequential access for the current zVariable (in the current CDF). Note that a current sequential value is maintained for each zVariable individually. Required arguments are as follows:

in: `long recNum`

Record number.

in: `long indices[]`

Dimension indices. `indices` is a 1-dimensional array containing one element per dimension. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but a place holder is necessary).

The required preselected objects/states are the current CDF and its current zVariable.

<SELECT_,zVARs_CACHESIZE_>

Selects the number of cache buffers to be used for all of the zVariable files (of the current CDF). This operation is not applicable to a single-file CDF. The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library in greater detail. Required arguments are as follows:

in: `long numBuffers`

The number of cache buffers to be used.

The only required preselected object/state is the current CDF.

<SELECT_,zVARs_RECNUMBER_>

Selects the current record number for each zVariable in the current CDF. This operation is provided to simplify the selection of the current record numbers for the zVariables involved in a multiple variable access operation (see the Concepts chapter in the CDF User's Guide). Required arguments are as follows:

in: `long recNum`

Record number.

The only required preselected object/state is the current CDF.

6.7 More Examples

Several more examples of the use of `CDFlib` follow. In each example it is assumed that the current CDF has already been selected (either implicitly by creating/opening the CDF or explicitly with `<SELECT_,CDF_>`).

6.7.1 rVariable Creation.

In this example an rVariable will be created with a pad value being specified; initial records will be written; and the number of records to extend the rVariable when necessary will be specified. Note that the pad value was specified before the initial records. This results in the specified pad value being written. Had the pad value not been specified first, the initial records would have been written with the default pad value. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
```

```
CDFstatus   status;                    /* Status returned from CDF
                                          library. */
long        dimVarys[2];               /* Dimension variances. */
long        varNum;                    /* rVariable number. */
float       padValue = -999.9;         /* Pad value. */
.
.
dimVarys[0] = VARY;
dimVarys[1] = VARY;
status = CDFlib (CREATE_, rVAR_, "HUMIDITY", CDF_REAL4, 1, VARY,
                                 dimVarys, &varNum,
                 PUT_, rVAR_PADVALUE_, &padValue,
                       rVAR_INITIALRECS_, (long) 500,
                       rVAR_EXTENDRECS_, (long) 50,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.2 zVariable Creation (Character String Data Type).

In this example a zVariable with a character string data type will be created with a pad value being specified. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;                    /* Status returned from CDF
                                          library. */
long        dimVarys[1];               /* Dimension variances. */
long        varNum;                    /* zVariable number. */
long        numDims = 1;               /* Number of dimensions. */
static long dimSizes[1] = { 20 };      /* Dimension sizes. */
long        numElems = 10;             /* Number of elements (characters
                                          in this case). */
static char padValue = "**********";   /* Pad value. */
.
.
dimVarys[0] = VARY;
status = CDFlib (CREATE_, zVAR_, "Station", CDF_CHAR, numElems, numDims,
                                 dimSizes, NOVARY, dimVarys, &varNum,
                 PUT_, zVAR_PADVALUE_, padValue,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.3 Hyper Read with Subsampling.

In this example an rVariable will be subsampled in a CDF whose rVariables are 2-dimensional and have dimension sizes `[100,200]`. The CDF is row major, and the data type of the rVariable is `CDF_UINT2`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;              /* Status returned from CDF library. */
unsigned short values[50][100];     /* Buffer to receive values. */
long           recCount = 1;        /* Record count, one record per hyper
                                       get. */
long           recInterval = 1;     /* Record interval, set to one to
                                       indicate contiguous records (really
                                       meaningless since record count is
                                       one). */
static long    indices[2] = {0,0};   /* Dimension indices, start each read
                                       at 0,0 of the array. */
static long    counts[2] = {50,100}; /* Dimension counts, half of the
                                       values along each dimension will
                                       be read. */
static long    intervals[2] = {2,2}; /* Dimension intervals, every other
                                       value along each dimension will be
                                       read. */
long           recNum;              /* Record number. */
long           maxRec;              /* Maximum rVariable record
                                       number in the CDF - this was
                                       determined with a call to
                                       CDFinquire. */
.
.
status = CDFlib (SELECT_, rVAR_NAME_, "BRIGHTNESS",
                          rVARs_RECCOUNT_, recCount,
                          rVARs_RECINTERVAL_, recInterval,
                          rVARs_DIMINDICES_, indices,
                          rVARs_DIMCOUNTS_, counts,
                          rVARs_DIMINTERVALS_, intervals,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);

for (recNum = 0; recNum <= maxRec; recNum++) {
   status = CDFlib (SELECT_, rVARs_RECNUMBER_, recNum,
                    GET_, rVAR_HYPERDATA_, values,
                    NULL_);
   if (status != CDF_OK) UserStatusHandler (status);
   .
   .
```

```
      /* process values */
      .
      .
   }
   .
   .
```

6.7.4 Attribute Renaming.

In this example the attribute named `Tmp` will be renamed `TMP`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus      status;               /* Status returned from CDF library. */
.
.
status = CDFlib (SELECT_, ATTR_NAME_, "Tmp",
                 PUT_, ATTR_NAME, "TMP",
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.5 Sequential Access.

In this example the values for a zVariable will be averaged. The values will be read using the sequential access method (see the Concepts chapter in the CDF User's Guide). Each value in each record will be read and averaged. It is assumed that the data type of the zVariable has been determined to be `CDF_REAL4`. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;              /* Status returned from CDF library. */
long        varNum;              /* zVariable number. */
long        recNum = 0;          /* Record number, start at first record. */
static long indices[2] = {0,0};  /* Dimension indices. */
float       value;               /* Value read. */
double      sum = 0.0;           /* Sum of all values. */
long        count = 0;           /* Number of values. */
```

```
float       ave;                /* Average value. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "FLUX", &varNum,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);

status = CDFlib (SELECT_, zVAR_, varNum,
                          zVAR_SEQPOS_, recNum, indices,
                 GET_, zVAR_SEQDATA_, &value,
                 NULL_);
while (status >= CDF_OK) {
  sum += value;
  count++;
  status = CDFlib (GET_, zVAR_SEQDATA_, &value,
                   NULL_);
}

if (status != END_OF_VAR) UserStatusHandler (status);

ave = sum / count;
.
.
```

6.7.6 Attribute rEntry Writes.

In this example a set of attribute rEntries for a particular rVariable will be written. It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus     status;                  /* Status returned from CDF
                                          library. */
static float  scale[2] = {-90.0,90.0}; /* Scale, minimum/maximum. */
.
.
status = CDFlib (SELECT_, rENTRY_NAME_, "LATITUDE",
                          ATTR_NAME_, "FIELDNAM",
                 PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                     "Latitude            ",
                 SELECT_, ATTR_NAME_, "SCALE",
                 PUT_, rENTRY_DATA_, CDF_REAL4, (long) 2, scale,
                 SELECT_, ATTR_NAME_, "UNITS",
                 PUT_, rENTRY_DATA_, CDF_CHAR, (long) 20,
                                     "Degrees north       ",
                 NULL_);
```

```
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.7 Multiple zVariable Write.

In this example full-physical records will be written to the zVariables in a CDF. Note the ordering of the zVariables (see the Concepts chapter in the CDF User's Guide). It is assumed that the current CDF has already been selected.

```
.
.
#include "cdf.h"
.
.
CDFstatus   status;             /* Status returned from CDF library. */
short       time;               /* 'Time' value. */
char        vectorA[3];         /* 'vectorA' values. */
double      vectorB[5];         /* 'vectorB' values. */
long        recNumber;          /* Record number. */
char        buffer[45];         /* Buffer of full-physical records. */
long        varNumbers[3];      /* Variable numbers. */
.
.
status = CDFlib (GET_, zVAR_NUMBER_, "vectorB", &varNumbers[0],
                       zVAR_NUMBER_, "time", &varNumbers[1],
                       zVAR_NUMBER_, "vectorA", &varNumbers[2],
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
for (recNumber = 0; recNumber < 100; recNumber++) {
   .
   /* read values from input file */
   .
   memmove (&buffer[0], vectorB, 40);
   memmove (&buffer[40], &time, 2);
   memmove (&buffer[42], vectorA, 3);
   status = CDFlib (SELECT_, zVARs_RECNUMBER_, recNumber,
                    PUT_, zVARs_RECDATA_, 3L, varNumbers, buffer,
                    NULL_);
  if (status != CDF_OK) UserStatusHandler (status);
}
.
.
```

Note that it would be more efficient to read the values directly into `buffer`. The method shown here was used to illustrate how to create the buffer of full-physical records.

6.8 A Potential Mistake We Don't Want You to Make

The following example illustrates one of the most common mistakes made when using the Internal Interface in a C application. Please don't do something like the following:

```
.
.
#include "cdf.h"
.
.
CDFid          id;                          /* CDF identifier (handle). */
CDFstatus      status;                      /* Status returned from CDF
                                               library. */
long           varNum;                      /* zVariable number. */
.
.
status = CDFlib (SELECT_, CDF_, id,
                 GET_, zVAR_NUMBER_, "EPOCH", &varNum,
                 SELECT_, zVAR_, varNum,                  /* ERROR! */
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

It looks like the current zVariable will be selected based on the zVariable number determined by using the `<GET_,zVAR_NUMBER_>` operation. What actually happens is that the zVariable number passed to the `<SELECT_,zVAR_>` operation is undefined. This is because the C compiler is passing **varNum** by value rather than reference.[6] Since the argument list passed to **CDFlib** is created before **CDFlib** is called, **varNum** does not yet have a value. Only after the `<GET_,zVAR_NUMBER_>` operation is performed does **varNum** have a valid value. But at that point it's too late since the argument list has already been created. Don't do this.

In this type of situation you would have to make two calls to **CDFlib**. The first would inquire the zVariable number and the second would select the current zVariable. Because the `<SELECT_,zVAR_NAME_>` operation allows you to select the current zVariable directly by name, this particular situation should never arise. It was simply used as an example. If you do encounter a situation like this that you think could be avoided with a new operation, contact CDFsupport; we're adding new operations all the time.

6.9 Custom C Functions

Most of the Standard Interface functions callable from C applications are implemented as C macros that call **CDFlib** (Internal Interface). For example, the **CDFcreate** function is actually defined as the following C macro:

```
#define CDFcreate(CDFpath,numDims,dimSizes,encoding,majority,id) \
CDFlib (CREATE_, CDF_, CDFpath, numDims, dimSizes, id, \
```

[6] Fortran programmers can get away with doing something like this because everything is passed by reference.

```
        PUT_, CDF_ENCODING_, encoding, \
              CDF_MAJORITY_, majority, \
        NULL_)
```

These macros are defined in `cdf.h`. Where your application calls `CDFcreate`, the C compiler (preprocessor) expands the macro into the corresponding call to `CDFlib`.

The flexibility of `CDFlib` allows you to define your own custom CDF functions using C macros. For instance, a function that inquires the format of a CDF could be defined as follows:

```
#define CDFinquireFormat(id,format) \
CDFlib (SELECT_, CDF_, id, \
        GET_, CDF_FORMAT_, format, \
        NULL_)
```

Your application would call the function as follows:

```
.
.
CDFid     id;            /* CDF identifier. */
CDFstatus status;        /* Returned status code. */
long      format;        /* Format of CDF. */
.
.
status = CDFinquireFormat (id, &format);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

Chapter 7

Interpreting CDF Status Codes

Most CDF functions return a status code of type `CDFstatus`. The symbolic names for these codes are defined in `cdf.h` and should be used in your applications rather than using the true numeric values. Appendix A explains each status code. When the status code returned from a CDF function is tested, the following rules apply.

`status > CDF_OK`	Indicates successful completion but some additional information is provided. These are informational codes.
`status = CDF_OK`	Indicates successful completion.
`CDF_WARN < status < CDF_OK`	Indicates that the function completed but probably not as expected. These are warning codes.
`status < CDF_WARN`	Indicates that the function did not complete. These are error codes.

The following example shows how you could check the status code returned from CDF functions.

```
CDFstatus status;
.
.
status = CDFfunction (...);   /* any CDF function returning CDFstatus */
if (status != CDF_OK) {
  UserStatusHandler (status, ...);
  .
  .
}
```

In your own status handler you can take whatever action is appropriate to the application. An example status handler follows. Note that no action is taken in the status handler if the status is `CDF_OK`.

```
#include <stdio.h>
```

```
   #include "cdf.h"

   void UserStatusHandler (status)
   CDFstatus status;
   {
   char message[CDF_STATUSTEXT_LEN+1];

   if (status < CDF_WARN) {
     printf ("An error has occurred, halting...\n");
     CDFerror (status, message);
     printf ("%s\n", message);
     exit (status);
   }
   else
     if (status < CDF_OK) {
       printf ("Warning, function may not have completed as expected...\n");
       CDFerror (status, message);
       printf ("%s\n", message);
     }
     else
       if (status > CDF_OK) {
         printf ("Function completed successfully, but be advised that...\n");
         CDFerror (status, message);
         printf ("%s\n", message);
       }
   return;
}
```

Explanations for all CDF status codes are available to your applications through the function `CDFerror`. `CDFerror` writes to a text string an explanation of a given status code.

Chapter 8

EPOCH Utility Routines

Several functions exist that compute, break down, parse, and encode CDF_EPOCH values. These functions may be called by applications using the CDF_EPOCH data type and are included in the CDF library. Function prototypes for these functions may be found in the include file cdf.h. The Concepts chapter in the CDF User's Guide describes EPOCH values.

8.1 computeEPOCH

computeEPOCH calculates a CDF_EPOCH value given the individual components. If an illegal component is detected, the value returned will be -1.0.

```
double computeEPOCH(     /* Out -- CDF_EPOCH value returned. */
long year,               /* In  -- Year (AD, e.g., 1994). */
long month,              /* In  -- Month (1-12). */
long day,                /* In  -- Day (1-31). */
long hour,               /* In  -- Hour (0-23). */
long minute,             /* In  -- Minute (0-59). */
long second,             /* In  -- Second (0-59). */
long msec);              /* In  -- Millisecond (0-999). */
```

8.2 EPOCHbreakdown

EPOCHbreakdown breaks down a CDF_EPOCH value into the individual components.

```
void EPOCHbreakdown(
double epoch,            /* In  -- The CDF_EPOCH value. */
long   *year,            /* Out -- Year (AD, e.g., 1994). */
long   *month,           /* Out -- Month (1-12). */
long   *day,             /* Out -- Day (1-31). */
```

```
long   *hour,           /* Out -- Hour (0-23). */
long   *minute,         /* Out -- Minute (0-59). */
long   *second,         /* Out -- Second (0-59). */
long   *msec);          /* Out -- Millisecond (0-999). */
```

8.3 parseEPOCH

`parseEPOCH` parses a standard date/time character string and returns a `CDF_EPOCH` value. The format of the string is defined in the Concepts chapter in the CDF User's Guide (and is also the format produced by the `encodeEPOCH` function). If an illegal field is detected in the string the value returned will be `-1.0`.

```
double parseEPOCH(                      /* Out -- CDF_EPOCH value
                                                  returned. */
char epString[EPOCH_STRING_LEN+1]);     /* In  -- The standard date/time
                                                  character string. */
```

`EPOCH_STRING_LEN` is defined in `cdf.h`.

8.4 encodeEPOCH

`encodeEPOCH` encodes a `CDF_EPOCH` value into a standard date/time character string. The format of the string is defined in the Concepts chapter in the CDF User's Guide (and is also the format expected by the `parseEPOCH` function).

```
void encodeEPOCH(
double epoch;                            /* In  -- the CDF_EPOCH value. */
char epString[EPOCH_STRING_LEN+1]);      /* Out -- The standard date/time
                                                   character string. */
```

`EPOCH_STRING_LEN` is defined in `cdf.h`.

8.5 encodeEPOCH1

`encodeEPOCH1` encodes a `CDF_EPOCH` value into an alternate date/time character string. The format of the string is `yyyymmdd.ttttttt`, where `yyyy` is the year, `mm` is the month (1-12), `dd` is the day of the month (1-31), and `ttttttt` is the fraction of the day (e.g., `5000000` is 12 o'clock noon).

```
void encodeEPOCH1(
double epoch;                            /* In  -- the CDF_EPOCH value. */
char epString[EPOCH1_STRING_LEN+1]);     /* Out -- The alternate date/time
                                                   character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

8.6 encodeEPOCH2

encodeEPOCH2 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyymoddhhmmss where yyyy is the year, mo is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), and ss is the second (0-59).

```
void encodeEPOCH2(
double epoch;                          /* In  -- the CDF_EPOCH value. */
char epString[EPOCH2_STRING_LEN+1]);   /* Out -- The alternate date/time
                                                 character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

8.7 encodeEPOCH3

encodeEPOCH3 encodes a CDF_EPOCH value into an alternate date/time character string. The format of the string is yyyy-mo-ddThh:mm:ss.cccZ where yyyy is the year, mo is the month (1-12), dd is the day of the month (1-31), hh is the hour (0-23), mm is the minute (0-59), ss is the second (0-59), and ccc is the millisecond (0-999).

```
void encodeEPOCH3(
double epoch;                          /* In  -- the CDF_EPOCH value. */
char epString[EPOCH3_STRING_LEN+1]);   /* Out -- The alternate date/time
                                                 character string. */
```

EPOCH3_STRING_LEN is defined in cdf.h.

Appendix A

Status Codes

A.1 Introduction

A status code is returned from most CDF functions. The `cdf.h` (for C) and `CDF.INC` (for Fortran) include files contain the numerical values (constants) for each of the status codes (and for any other constants referred to in the explanations). The CDF library Standard Interface functions `CDFerror` (for C) and `CDF_error` (for Fortran) can be used within a program to inquire the explanation text for a given status code. The Internal Interface can also be used to inquire explanation text.

There are three classes of status codes: informational, warning, and error. The purpose of each is as follows:

Informational	Indicates success but provides some additional information that may be of interest to an application.
Warning	Indicates that the function completed but possibly not as expected.
Error	Indicates that a fatal error occurred and the function aborted.

Status codes fall into classes as follows:

Error codes < `CDF_WARN` < Warning codes < `CDF_OK` < Informational codes

`CDF_OK` indicates an unqualified success (it should be the most commonly returned status code). `CDF_WARN` is simply used to distinguish between warning and error status codes.

A.2 Status Codes and Messages

The following list contains an explanation for each possible status code. Whether a particular status code is considered informational, a warning, or an error is also indicated.

`ATTR_EXISTS`	Named attribute already exists — cannot create or rename. Each attribute in a CDF must have a unique name. Note that trailing blanks are ignored by the CDF library when comparing attribute names. [Error]
`ATTR_NAME_TRUNC`	Attribute name truncated to `CDF_ATTR_NAME_LEN` characters. The attribute was created but with a truncated name. [Warning]
`BAD_ALLOCATE_RECS`	An illegal number of records to allocate for a variable was specified. For RV variables the number must be one or greater. For NRV variables the number must be exactly one. [Error]
`BAD_ARGUMENT`	An illegal/undefined argument was passed. Check that all arguments are properly declared and initialized. [Error]
`BAD_ATTR_NAME`	Illegal attribute name specified. Attribute names must contain at least one character, and each character must be printable. [Error]
`BAD_ATTR_NUM`	Illegal attribute number specified. Attribute numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]
`BAD_CACHESIZE`	An illegal number of cache buffers was specified. The value must be at least one (1). [Error]
`BAD_CDF_EXTENSION`	An illegal file extension was specified for a CDF. In general, do not specify an extension except possibly for a single-file CDF which has been renamed with a different file extension or no file extension. [Error]
`BAD_CDF_ID`	CDF identifier is unknown or invalid. The CDF identifier specified is not for a currently open CDF. [Error]
`BAD_CDF_NAME`	Illegal CDF name specified. CDF names must contain at least one character, and each character must be printable. Trailing blanks are allowed but will be ignored. [Error]
`BAD_CDFSTATUS`	Unknown CDF status code received. The status code specified is not used by the CDF library. [Error]
`BAD_DATA_TYPE`	Unknown data type specified. The CDF data types are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_DECODING`	Unknown decoding specified. The CDF decodings are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_DIM_COUNT`	Illegal dimension count specified. A dimension count must be at least one (1) and not greater than the size of the dimension. [Error]
`BAD_DIM_INDEX`	One or more dimension index is out of range. A valid value must be specified regardless of the dimension variance. Note also that the combination of dimension index, count, and in-

	terval must not specify an element beyond the end of the dimension. [Error]
`BAD_DIM_INTERVAL`	Illegal dimension interval specified. Dimension intervals must be at least one (1). [Error]
`BAD_DIM_SIZE`	Illegal dimension size specified. A dimension size must be at least one (1). [Error]
`BAD_ENCODING`	Unknown data encoding specified. The CDF encodings are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_ENTRY_NUM`	Illegal attribute entry number specified. Entry numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. [Error]
`BAD_EXTEND_RECS`	An illegal number of extend records has been specified. The number of extend records must be at least zero (0). A value of zero indicates that the default should be used. Note that extend records are only significant for variables that are not NRV and are in single-file CDFs. [Error]
`BAD_FNC_OR_ITEM`	The specified function or item is illegal. Check that the proper number of arguments are specified for each operation being performed. Also make sure that `NULL_` is specified as the last operation. [Error]
`BAD_FORMAT`	Unknown format specified. The CDF formats are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_INITIAL_RECS`	An illegal number of records to initially write has been specified. The number of initial records must be at least one (1). [Error]
`BAD_MAJORITY`	Unknown variable majority specified. The CDF variable majorities are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_MALLOC`	Unable to allocate dynamic memory — system limit reached. Contact CDF User Support if this error occurs. [Error]
`BAD_NEGtoPOSfp0_MODE`	An illegal `-0.0` to `0.0` mode was specified. The `-0.0` to `0.0` modes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_NUM_DIMS`	The number of dimensions specified is out of the allowed range. Zero (0) through `CDF_MAX_DIMS` dimensions are allowed. If more are needed, contact CDF User Support. [Error]
`BAD_NUM_ELEMS`	The number of elements of the data type is illegal. The number of elements must be at least one (1). For non-character string variables, the number of elements must always be one (1). [Error]