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 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_ALLOCATEBLOCK_>`

Specifies a range of records to allocate for the current rVariable (in the current CDF). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long firstRecord`

The first record number to allocate.

in: `long lastRecord`

The last record number to allocate.

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

<PUT_,rVAR_ALLOCATERECS_>

Specifies the number of records to allocate for the current rVariable (in the current CDF). The records are allocated beginning at record number 0 (zero). This operation is only applicable to uncompressed rVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long nRecords`

Number of records to allocate.

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

<PUT_,rVAR_BLOCKINGFACTOR_>[7]

Specifies the blocking factor for the current rVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes a variable's blocking factor. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs. Required arguments are as follows:

in: `long blockingFactor`

The blocking factor. A value of zero (0) indicates that the default blocking factor should be used.

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

<PUT_,rVAR_COMPRESSION_>

Specifies the compression type/parameters for the current rVariable (in current CDF). Required arguments are as follows:

in: `long cType`

The compression type. The types of compressions are described in Section 4.10.

in: `long cParms[]`

The compression parameters. The compression parameters are described in Section 4.10.

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

[7]The item `rVAR_BLOCKINGFACTOR_` was previously named `rVAR_EXTENDRECS_`.

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 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 data types this must be one (1) — arrays of values are not allowed for non-character 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. 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_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). The records are written beginning at record number `0` (zero). 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 nRecords`

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_,rVAR_SPARSEARRAYS_>

Specifies the sparse arrays type/parameters for the current rVariable (in the current CDF). Required arguments are as follows:

in: `long sArraysType`

The sparse arrays type. The types of sparse arrays are described in Section 4.11.

in: `long sArraysParms[]`

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.

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

`<PUT_,rVAR_SPARSERECORDS_>`

Specifies the sparse records type for the current rVariable (in the current CDF). Required arguments are as follows:

in: `long sRecordsType`

The sparse records type. The types of sparse records are described in Section 4.11.

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

`<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 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_ALLOCATEBLOCK_>`

Specifies a range of records to allocate for the current zVariable (in the current CDF). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long firstRecord`

The first record number to allocate.

in: `long lastRecord`

The last record number to allocate.

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

`<PUT_,zVAR_ALLOCATERECS_>`

Specifies the number of records to allocate for the current zVariable (in the current CDF). The records are allocated beginning at record number `0` (zero). This operation is only applicable to uncompressed zVariables in single-file CDFs. The Concepts chapter in the CDF User's Guide describes the allocation of variable records. Required arguments are as follows:

in: `long nRecords`

Number of records to allocate.

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

`<PUT_,zVAR_BLOCKINGFACTOR_>`[8]

Specifies the blocking factor for the current zVariable (in the current CDF). The Concepts chapter in the CDF User's Guide describes a variable's blocking factor. **NOTE:** The blocking factor has no effect for NRV variables or multi-file CDFs. Required arguments are as follows:

in: `long blockingFactor`

The blocking factor. A value of zero (0) indicates that the default blocking factor should be used.

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

`<PUT_,zVAR_COMPRESSION_>`

Specifies the compression type/parameters for the current zVariable (in current CDF). Required arguments are as follows:

in: `long cType`

The compression type. The types of compressions are described in Section 4.10.

in: `long cParms[]`

The compression parameters. The compression parameters are described in Section 4.10.

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

[8] The item `zVAR_BLOCKINGFACTOR_` was previously named `zVAR_EXTENDRECS_`.

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 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 data types this must be one (1) — arrays of values are not allowed for non-character 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. 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_INITIALRECS_>`

Specifies the number of records to initially write to the current zVariable (in the current CDF). The records are written beginning at record number `0` (zero). 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 nRecords`

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_,zVAR_SPARSEARRAYS_>`

Specifies the sparse arrays type/parameters for the current zVariable (in the current CDF). Required arguments are as follows:

in: `long sArraysType`

The sparse arrays type. The types of sparse arrays are described in Section 4.11.

in: `long sArraysParms[]`

The sparse arrays parameters. The sparse arrays parameters are described in Section 4.11.

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

`<PUT_,zVAR_SPARSERECORDS_>`

Specifies the sparse records type for the current zVariable (in the current CDF). Required arguments are as follows:

in: `long sRecordsType`

The sparse records type. The types of sparse records are described in Section 4.11.

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

`<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. **NOTE:** Selecting the current attribute by number (see <SELECT_,ATTR_>) is more efficient. 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 dotCDF file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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 4.15.

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 4.13.

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

`<SELECT_,CDF_SCRATCHDIR_>`

Selects a directory to be used for scratch files (by the CDF library) for the current CDF. The Concepts chapter in the CDF User's Guide describes how the CDF library uses scratch files. This scratch directory will override the directory specified by the the `CDF$TMP` logical name (on VMS systems) or `CDF_TMP` environment variable (on UNIX and MS-DOS systems). Required arguments are as follows:

in: `char *scratchDir`

The directory to be used for scratch files. The length of this directory specification is limited only by the operating system being used.

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 4.14.

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

`<SELECT_,COMPRESS_CACHESIZE_>`

Selects the number of cache buffers to be used for the compression scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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_,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.) **NOTE:** Selecting the current rEntry by number (see `<SELECT_,rENTRY_>`) is more efficient. 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. 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. **NOTE:** Selecting the current rVariable by number (see `<SELECT_,rVAR_>`) is more efficient. 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_RESERVEPERCENT_>`

Selects the reserve percentage to be used for the current rVariable (in the current CDF). This operation is only applicable to compressed rVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

in: `long percent`

The reserve percentage.

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

`<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. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional rVariables this argument is ignored (but must be present).

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. 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. 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. 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. 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_,STAGE_CACHESIZE_>`

Selects the number of cache buffers to be used for the staging scratch file (for the current CDF). The Concepts chapter in the CDF User's Guide describes the caching scheme used by the CDF library. 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_,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.) **NOTE:** Selecting the current zEntry by number (see `<SELECT_,zENTRY_>`) is more efficient. 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. 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. 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. 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. 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. **NOTE:** Selecting the current zVariable by number (see <SELECT_,zVAR_>) is more efficient. 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_RESERVEPERCENT_>

Selects the reserve percentage to be used for the current zVariable (in the current CDF). This operation is only applicable to compressed zVariables. The Concepts chapter in the CDF User's Guide describes the reserve percentage scheme used by the CDF library. Required arguments are as follows:

in: `long percent`

The reserve percentage.

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. Each element of `indices` specifies the corresponding dimension index. For 0-dimensional zVariables this argument is ignored (but must be present).

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. 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 rVariable's blocking factor 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_BLOCKINGFACTOR_, (long) 50,
                 NULL_);
if (status != CDF_OK) UserStatusHandler (status);
.
.
```

6.7.2 zVariable Creation (Character Data Type).

In this example a zVariable with a character 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 to `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.[9] 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. 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.

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(CDFname,numDims,dimSizes,encoding,majority,id) \
CDFlib (CREATE_, CDF_, CDFname, numDims, dimSizes, id, \
        PUT_, CDF_ENCODING_, encoding, \
              CDF_MAJORITY_, majority, \
        NULL_)
```

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

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` encodes in a text string an explanation of a given status code.

Chapter 8

EPOCH Utility Routines

Several functions exist that compute, decompose, 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). */
```

NOTE: There are two variations on how computeEPOCH may be used. If the month argument is 0 (zero), then the day argument is assumed to be the day of the year (DOY) having a range of 1 through 366. Also, if the hour, minute, and second arguments are all 0 (zero), then the msec argument is assumed to be the millisecond of the day having a range of 0 through 86400000.

8.2 EPOCHbreakdown

EPOCHbreakdown decomposes 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 encodeEPOCH

encodeEPOCH encodes a CDF_EPOCH value into the standard date/time character string. The format of the string is dd-mmm-yyyy hh:mm:ss.ccc where dd is the day of the month (1-31), mmm is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec), yyyy is the year, 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 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.

8.4 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.

8.5 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.

8.6 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.

8.7 encodeEPOCHx

encodeEPOCHx encodes a CDF_EPOCH value into a custom date/time character string. The format of the encoded string is specified by a format string.

```
void encodeEPOCHx(
double epoch;                          /* In  -- The CDF_EPOCH value. */
char format[EPOCHx_FORMAT_MAX];        /* In  ---The format string. */
char encoded[EPOCHx_STRING_MAX]);      /* Out -- The custom date/time
                                                 character string. */
```

The format string consists of EPOCH components which are encoded and text which is simply copied to the encoded custom string. Components are enclosed in angle brackets and consist of a component token and an optional width. The syntax of a component is: <token[.width]>. If the optional width contains a leading zero, then the component will be encoded with leading zeroes (rather than leading blanks).

The supported component tokens and their default widths are as follows. . .

Token	Meaning	Default
`dom`	Day of month (1-31)	`<dom.0>`
`doy`	Day of year (001-366)	`<doy.03>`
`month`	Month ('Jan','Feb',...,'Dec')	`<month>`
`mm`	Month (1,2,...,12)	`<mm.0>`
`year`	Year (4-digit)	`<year.04>`
`yr`	Year (2-digit)	`<yr.02>`
`hour`	Hour (00-23)	`<hour.02>`
`min`	Minute (00-59)	`<min.02>`
`sec`	Second (00-59)	`<sec.02>`
`fos`	Fraction of second.	`<fos.3>`
`fod`	Fraction of day.	`<fod.8>`

Note that a width of zero indicates that as many digits as necessary should be used to encoded the component. The `<month>` component is always encoded with three characters. The `<fos>` and `<fod>` components are always encoded with leading zeroes.

If a left angle bracket is desired in the encoded string, then simply specify two left angle brackets (`<<`) in the format string (character stuffing).

For example, the format string used to encode the standard EPOCH date/time character string (see Section 8.3) would be...

```
<dom.02>-<month>-<year> <hour>:<min>:<sec>.<fos>
```

`EPOCHx_FORMAT_LEN` and `EPOCHx_STRING_MAX` are defined in `cdf.h`.

8.8 parseEPOCH

`parseEPOCH` parses a standard date/time character string and returns a `CDF_EPOCH` value. The format of the string is that produced by the `encodeEPOCH` function described in Section 8.3. 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`.

8.9 parseEPOCH1

`parseEPOCH1` parses an alternate date/time character string and returns a `CDF_EPOCH` value. The format of the string is that produced by the `encodeEPOCH1` function described in Section 8.4. If an illegal field is detected in the string the value returned will be `-1.0`.

```
double parseEPOCH1(                 /* Out -- CDF_EPOCH value
                                               returned. */
char epString[EPOCH1_STRING_LEN+1]);   /* In  -- The alternate date/time
                                               character string. */
```

EPOCH1_STRING_LEN is defined in cdf.h.

8.10 parseEPOCH2

parseEPOCH2 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH2 function described in Section 8.5. If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH2(                 /* Out -- CDF_EPOCH value
                                               returned. */
char epString[EPOCH2_STRING_LEN+1]);   /* In  -- The alternate date/time
                                               character string. */
```

EPOCH2_STRING_LEN is defined in cdf.h.

8.11 parseEPOCH3

parseEPOCH3 parses an alternate date/time character string and returns a CDF_EPOCH value. The format of the string is that produced by the encodeEPOCH3 function described in Section 8.6. If an illegal field is detected in the string the value returned will be -1.0.

```
double parseEPOCH3(                 /* Out -- CDF_EPOCH value
                                               returned. */
char epString[EPOCH3_STRING_LEN+1]);   /* In  -- 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_BLOCKING_FACTOR`[1]	An illegal blocking factor was specified. Blocking factors must be at least zero (0). [Error]
`BAD_CACHESIZE`	An illegal number of cache buffers was specified. The value must be at least zero (0). [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_COMPRESSION_PARM`	An illegal compression parameter was specified. [Error]
`BAD_DATA_TYPE`	An unknown data type was specified or encountered. The CDF data types are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_DECODING`	An unknown decoding was 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.

[1] The status code `BAD_BLOCKING_FACTOR` was previously named `BAD_EXTEND_RECS`.

	[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 interval 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_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 variables with a non-character data type, the number of elements must always be one (1). [Error]
`BAD_NUM_VARS`	Illegal number of variables in a record access operation. [Error]

`BAD_READONLY_MODE`	Illegal read-only mode specified. The CDF read-only modes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_REC_COUNT`	Illegal record count specified. A record count must be at least one (1). [Error]
`BAD_REC_INTERVAL`	Illegal record interval specified. A record interval must be at least one (1). [Error]
`BAD_REC_NUM`	Record number is out of range. Record numbers must be at least zero (0) for C applications and at least one (1) for Fortran applications. Note that a valid value must be specified regardless of the record variance. [Error]
`BAD_SCOPE`	Unknown attribute scope specified. The attribute scopes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`BAD_SCRATCH_DIR`	An illegal scratch directory was specified. The scratch directory must be writeable and accessable (if a relative path was specified) from the directory in which the application has been executed. [Error]
`BAD_SPARSEARRAYS_PARM`	An illegal sparse arrays parameter was specified. [Error]
`BAD_VAR_NAME`	Illegal variable name specified. Variable names must contain at least one character and each character must be printable. [Error]
`BAD_VAR_NUM`	Illegal variable number specified. Variable numbers must be zero (0) or greater for C applications and one (1) or greater for Fortran applications. [Error]
`BAD_zMODE`	Illegal zMode specified. The CDF zModes are defined in `cdf.h` for C applications and in `cdf.inc` for Fortran applications. [Error]
`CANNOT_ALLOCATE_RECORDS`	Records cannot be allocated for the given type of variable (e.g., a compressed variable). [Error]
`CANNOT_CHANGE`	Because of dependencies on the value, it cannot be changed. Some possible causes of this error follow: 1. Changing a CDF's data encoding after a variable value (including a pad value) or an attribute entry has been written. 2. Changing a CDF's format after a variable has been created or if a compressed single-file CDF. 3. Changing a CDF's variable majority after a variable value (excluding a pad value) has been written. 4. Changing a variable's data specification after a value (including the pad value) has been written to

that variable or after records have been allocated for that variable.

5. Changing a variable's record variance after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.
6. Changing a variable's dimension variances after a value (excluding the pad value) has been written to that variable or after records have been allocated for that variable.
7. Writing "initial" records to a variable after a value (excluding the pad value) has already been written to that variable.
8. Changing a variable's blocking factor when a compressed variable and a value (excluding the pad value) has been written or when a variable with sparse records and a value has been accessed.
9. Changing an attribute entry's data specification where the new specification is not equivalent to the old specification.

`CANNOT_COMPRESS`	The CDF or variable cannot be compressed. For CDFs, this occurs if the CDF has the multi-file format. For variables, this occurs if the variable is in a multi-file CDF, values have been written to the variable, or if sparse arrays have already been specified for the variable. [Error]
`CANNOT_SPARSEARRAYS`	Sparse arrays cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, records have been allocated for the variable, or if compression has already been specified for the variable. [Error]
`CANNOT_SPARSERECORDS`	Sparse records cannot be specified for the variable. This occurs if the variable is in a multi-file CDF, values have been written to the variable, or records have been allocated for the variable. [Error]
`CDF_CLOSE_ERROR`	Error detected while trying to close CDF. Check that sufficient disk space exists for the dotCDF file and that it has not been corrupted. [Error]
`CDF_CREATE_ERROR`	Cannot create the CDF specified — error from file system. Make sure sure that sufficient privilege exists to create the dotCDF file in the disk/directory location specified and that an open file quota has not already been reached. [Error]
`CDF_DELETE_ERROR`	Cannot delete the CDF specified — error from file system. Unsufficient privileges exist the delete the CDF file(s). [Error]

CDF_EXISTS	The CDF named already exists — cannot create it. The CDF library will not overwrite an existing CDF. [Error]
CDF_INTERNAL_ERROR	An unexpected condition has occurred in the CDF library. Report this error to CDFsupport. [Error]
CDF_NAME_TRUNC	CDF pathname truncated to CDF_PATHNAME_LEN characters. The CDF was created but with a truncated name. [Warning]
CDF_OK	Function completed successfully.
CDF_OPEN_ERROR	Cannot open the CDF specified — error from file system. Check that the dotCDF file is not corrupted and that sufficient privilege exists to open it. Also check that an open file quota has not already been reached. [Error]
CDF_READ_ERROR	Failed to read the CDF file — error from file system. Check that the dotCDF file is not corrupted. [Error]
CDF_WRITE_ERROR	Failed to write the CDF file — error from file system. Check that the dotCDF file is not corrupted. [Error]
COMPRESSION_ERROR	An error occured while compressing a CDF or block of variable records. This is an internal error in the CDF library. Contact CDF User Support. [Error]
CORRUPTED_V2_CDF	This Version 2 CDF is corrupted. An error has been detected in the CDF's control information. If the CDF file(s) are known to be valid, please contact CDF User Support. [Error]
DECOMPRESSION_ERROR	An error occured while decompressing a CDF or block of variable records. The most likely cause is a corrupted dotCDF file. [Error]
DID_NOT_COMPRESS	For a compressed variable, a block of records did not compress to smaller than their uncompressed size. They have been stored uncompressed. This can result if the blocking factor is set too low or if the characteristics of the data are such that the compression algorithm choosen is unsuitable. [Informational]
EMPTY_COMPRESSED_CDF	The compressed CDF being opened is empty. This will result if a program which was creating/modifying the CDF abnormally terminated. [Error]
END_OF_VAR	The sequential access current value is at the end of the variable. Reading beyond the end of the last physical value for a variable is not allowed (when performing sequential access). [Error]
FORCED_PARAMETER	A specified parameter was forced to an acceptable value (rather than an error being returned). [Warning]
IBM_PC_OVERFLOW	An operation involving a buffer greater than 64k bytes in size has been specified for PCs running 16-bit DOS/Windows 3.*. [Error]
ILLEGAL_FOR_SCOPE	The operation is illegal for the attribute's scope. For example, only gEntries may be written for gAttributes — not rEntries