COASTWATCH FORMAT SOFTWARE LIBRARY AND UTILITIES: USER'S GUIDE

VERSION 2
REVISED NOVEMBER 1999

Prepared by:

Peter Hollemans
CoastWatch Operations Manager
West Coast Regional Node

U.S. DEPARTMENT OF COMMERCE
NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION
NATIONAL MARINE FISHERIES SERVICE
SOUTHWEST FISHERIES SCIENCE CENTER
8604 LA JOLLA SHORES DR.
LA JOLLA, CA 92037-1508

CoastWatch Contacts:

Mail questions, comments, suggestions, and bug reports to:

CoastWatch Operations Manager
CoastWatch West Coast Regional Node
P.O. Box 271
La Jolla, CA 92038-0271, USA

Send electronic correspondence to:

`cwatch@cwatchwc.ucsd.edu`

Internet Access:

The CoastWatch Format (CWF) Software Library and Utilities distribution is available from the CoastWatch West Coast Regional Node web site:

`http://cwatchwc.ucsd.edu`

Contents

1 INTRODUCTION

1.1 History

The original need for a set of standard routines to read and write CoastWatch data files arose at the CoastWatch West Coast Regional Node in La Jolla, CA, in September, 1997. A software change was planned for early 1998, from using the Global Imaging GAE image processing package to the SeaSpace Corporation TeraScan package. In doing so, a new routine was needed to translate TeraScan data format (TDF) into CoastWatch format (CWF[1]). Although a routine existed to translate GAE files into CWF, it was not possible to adapt it for handling TDF files. In addition to simple translation, more sophisticated handling of CWF files was desired above and beyond what could be provided by the existing software packages (DECCON, IMGMAP, WIM, CCoast, etc.). In short, there existed no coherent application programming interface (API) for CWF files with a standard set of utilities supported on multiple PC and Unix computer platforms.

In October, 1997, an original CWF routine library (Version 1) was written in ANSI-C. That library was used to write a TDF to CWF translation routine and some simple CWF utilities. Some obvious limitations of the library spurred a refinement of the code to produce Version 2. In particular, Version 1:

- accessed all the data at once - no good for low-memory DOS machines
- did not handle angle and cloud mask files
- had an interface that is very IMGMAP specific

Version 2 has eliminated these problems and added other improvements. The current library and utilities source code and documentation is available from the CoastWatch West Coast Regional Node web site:

`http://cwatchwc.ucsd.edu`

1.2 NetCDF Heritage

The Version 2 CWF library is based on the netCDF data model presented in [8]. Although behind the scenes the file format is actually the standard CoastWatch format, the interface routines mimic those of the netCDF library. In most cases, the routine names and argument types are carbon-copies of the corresponding netCDF routines. A pseudo-netCDF interface was chosen because its functionality is well-suited for accessing CWF data, and should ease any future migrations to a hierarchical-type file format.

[1]Strictly speaking, *CoastWatch format* is simply a subset of the Image Mapping system format (IMGMAP), a standard for environmental image products developed by NOAA/NESDIS. For the purposes of this guide, however, the format will be referred to as CWF to dispell any confusion in the future when and if IMGMAP is abandoned as the CoastWatch data format.

Following with netCDF, the data in CWF files is accessed in terms of *dimensions*, *variables*, and *attributes*[2]. Unlike netCDF, only certain dimension, variable, and attribute names are allowed and there are restrictions on variable types and attribute types and values. Other limitations also exists, and will be explained in later sections.

1.3 Using the CWF Library

From an application programming standpoint, using the CWF library is simple. In general cases, only a small subset of all routines in the library need to be called to perform a specific task. In the following sections, templates for common sequences of CWF library calls are given for common tasks. Note that for clarity, the type-specific suffixes of routine names are omitted, "..." is inserted for an arbitrary sequence of statements, and routines that would be called multiple times are indented.

1.3.1 Creating a CWF Dataset

Following is a typical sequence of calls to create a new CWF dataset.

```
cw_create           /* create CWF dataset: enter define mode */
    ...
  cw_def_dim        /* define dimensions: from name and length */
    ...
  cw_def_var        /* define variables: from name, type, ... */
    ...
  cw_put_att        /* put attributes: assign attribute values */
    ...
cw_enddef           /* end definitions: leave define mode */
    ...
  cw_put_var        /* provide values for variables */
    ...
cw_close            /* close: save new CWF dataset */
```

Upon creating a CWF dataset, it will be in the first of two CWF *modes*. When accessing a CWF dataset, it is either in *define mode* or *data mode*. In define mode, you can create dimensions, attributes, and variables but you cannot read or write data. In data mode, you can access data and change attribute values, but not define new variables, dimensions, or attributes. Calling `cw_enddef` will end define mode and enter data mode. Once in data mode, you can write data values, change old values, and change the values of attributes. Finally, `cw_close` must explicitly be called for all datasets that are open for writing to make sure that data values are written correctly to disk before exiting.

[2]The reader should refer to [8] for a full explanation of the netCDF data model and routine behaviour. This guide only gives details on the limited CWF implementation of netCDF.

1.3.2 Reading a CWF Dataset

In contrast to creating a new CWF dataset, reading an existing CWF dataset is somewhat simpler. The following sequence of library calls might be used:

```
cw_open                 /* open existing CWF dataset */
    ...
  cw_inq_dimid          /* get dimension IDs */
    ...
  cw_inq_varid          /* get variable IDs */
    ...
  cw_get_att            /* get attribute values */
    ...
  cw_get_var            /* get values of variables */
    ...
cw_close                /* close CWF dataset */
```

After opening the dataset, `cw_inq_dimid` gets the dimension ID from the dimension name. Similarly, variable IDs are retrieved from the variable names. Then variable attributes and values can be obtained using the variable ID. Finally, the dataset is closed with `cw_close`. Note that for datasets opened for reading only, explicitly closing the dataset is not necessary.

1.3.3 Error Handling

Error handling in the CWF library is handled by way of an integer return status from any of the functions. If the return status indicates an error, the function `cw_strerror` may be used to translate the error into an error message string. For simplicity, the examples in this guide check the return status and call a separate function to handle the error.

1.4 CWF Data

This section discusses the four primitive CWF external data types, and the types of data access supported by the CWF interface.

1.4.1 CWF External Data Types

The external data types supported by the CWF interface are as follows:

char	8-bit characters intended for representing text
byte	8-bit unsigned integers
short	16-bit signed integers
float	11 or 16-bit scaled floating-point

These types were chosen to cover the range of values for variables and attributes encoded in a CWF file. They are called *external* because they correspond to different methods used by the interface for accessing the dataset on disk. When a program reads external CWF data into an internal variable, the data is converted to the internal type. This feature can be used to simplify code by using a sufficiently wide internal type to access a number of different external types. Only narrow to wide conversions are supported; for example, external byte data can be read into a floating-point internal variable, but external floating-point data cannot be read into an internal byte variable (type `unsigned char` in C). These conversion rules apply in a similar way to writing data. The one external type that the conversion rules do not apply to is **char** data. Only character data representing text strings can be written to or read from **char**.

1.4.2 Data Access

Data access in the CWF interface is limited to access by *array section*. An array section is a rectangular block of data specified by two vectors. The *start vector* gives the indices of the element in the corner closest to the origin. The *count vector* gives the lengths of the edges of the rectangle along each of the variable's dimensions, in order. The number of values accessed is the product of these edge lengths.

As an example, consider a CWF dataset with dimensions:

(rows, columns) = (512, 512)

To access the full block of data, the following would be used:

start = (0, 0)
count = (512, 512)

or to access row 100 only:

start = (100, 0)
count = (1, 512)

Note that indexing uses the C convention of starting at 0 rather than the FORTRAN convention of starting at 1. Section 4 “Variables” describes in detail the array section access routines.

2 DATASETS

This section presents the interfaces of the CWF functions that deal with CWF datasets as a whole, as well as the error status function. To access a CWF dataset, it is first referred to by file name. After opening, it is referred to by its *CWF ID*, a small non-negative integer. The CWF ID is like a logical unit number in FORTRAN or a file descriptor in C. After closing, the ID is no longer associated with the dataset.

The operations supported on a CWF dataset include:

- Create, given dataset name and whether to overwrite or not.
- Open, given dataset name and read/write mode.
- End define mode.
- Close, writing all data to disk.

2.1 Get error message string from error status: `cw_strerror`

The function `cw_strerror` returns a static reference to an error message string corresponding to an integer CWF error status.

Usage

`const char *cw_strerror (int cwerr);`

`cwerr`	An error status returned from a previous call to some CWF function.

Errors

`cw_strerr` returns a string indicating that the error is unknown if given an error status that does not correspond to any CWF error message.

Example

The following is a simple error handling function that uses `cw_strerror`:

```
#include <stdio.h>
#include <cwf.h>
    ...
void handle_error (int status) {
  if (status != CW_NOERR) {
    fprintf (stderr, "%s\n", cw_strerror (status));
    exit (-1);
  }
}
```

2.2 Create a CWF Dataset: `cw_create`

The creation function creates a new CWF dataset, returning the CWF ID. A creation mode specifies whether or not to overwrite an existing dataset of the same name. The new CWF dataset is placed in define mode, ready for dimension, variable, and attribute definitions.

Usage

```
int cw_create (const char *path, int cmode, int *cwidp);
```

`path`	The file name of the new CWF dataset.
`cmode`	The creation mode. A zero value (or `CW_CLOBBER`) specifies that a dataset of the same name should be overwritten. If `CW_NOCLOBBER` is specified, an existing dataset will not be overwritten; an error is returned if the dataset exists.
`cwidp`	Pointer to location where the returned CWF ID is to be stored.

Errors

`cw_create` returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The CWF dataset name contains a directory that does not exist, or you don't have access to.
- The dataset already exists and `CW_NOCLOBBER` was specified.
- The creation mode is invalid.

Example

The following code fragment creates a new CWF dataset called `w9813021.lc4`; the dataset is created only if the file does not already exist:

```
#include <cwf.h>
    ...
int status;
int cwid;
    ...
status = cw_create ("w9813021.lc4", CW_NOCLOBBER, &cwid);
if (status != CW_NOERR) handle_error (status);
```

2.3 Open a CWF Dataset for Access: `cw_open`

The function `cw_open` opens an existing CWF dataset for access.

Usage

```
int cw_open (const char *path, int omode, int *cwidp);
```

path	The file name of the CWF dataset to be opened.
omode	The open mode. A zero value (or CW_NOWRITE) specifies that the dataset by opened for read-only access. Specifying CW_WRITE opens the dataset as read-write - variable data and attribute values can be changed.
cwidp	Pointer to location where the returned CWF ID is to be stored.

Errors

cw_open returns the value CW_NOERR if no errors occurred. Possible causes of errors include:

- The CWF dataset does not exist.
- The open mode is invalid.
- The dataset contains variable data of a type not supported by the library.

Example

The following code opens the CWF dataset w9813021.lc4 for read-only access:

```
#include <cwf.h>
    ...
int status;
int cwid;
    ...
status = cw_open ("w9813021.lc4", CW_NOWRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
```

2.4 Leave Define Mode: cw_enddef

The function cw_enddef takes an open CWF dataset out of define mode. The definitions made while in define mode are checked and variable data initialized to a fill value (see Section 4.6 "Invalid Data Values"). The dataset is then placed in data mode and variable data can be read or written.

Usage

```
int cw_enddef (int cwid);
```

cwid	CWF ID, from a previous call to cw_create.

Errors

cw_enddef returns the value CW_NOERR if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The specified CWF dataset is not in define mode.
- No dimensions and/or variables were defined while in define mode.

Example

The following example uses `cw_enddef` to end define mode after creating a new CWF dataset:

```
#include <cwf.h>
    ...
int status;
int cwid;
    ...
status = cw_create ("w9813021.lc4", CW_NOCLOBBER, &cwid);
if (status != CW_NOERR) handle_error (status);

    ...             /* create dimensions, variables, attributes */

status = cw_enddef (cwid);  /* leave define mode */
if (status != CW_NOERR) handle_error (status);
```

2.5 Close an Open CWF Dataset: `cw_close`

The function `cw_close` will close an open CWF dataset. If the dataset is in define mode, `cw_enddef` will be called before closing. After an open CWF dataset is closed, its CWF ID may be reassigned to the next CWF dataset that is opened or created.

Usage

```
int cw_close (int cwid);
```

`cwid`	CWF ID, from a previous call to `cw_open` or `cw_create`.

Errors

`cw_close` returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- If in define mode, the automatic call to `cw_enddef` failed.
- If file compression was turned on and the file failed to compress correctly.

Example

The following example uses `cw_close` after creating a new CWF dataset:

```
#include <cwf.h>
    ...
int status;
int cwid;
    ...
status = cw_create ("w9813021.lc4", CW_NOCLOBBER, &cwid);
if (status != CW_NOERR) handle_error (status);

    ...               /* create dimensions, variables, attributes */

status = cw_close (cwid);   /* close CWF dataset */
if (status != CW_NOERR) handle_error (status);
```

3 DIMENSIONS

Dimensions for a CWF dataset are defined when it is created, while the CWF file is in define mode. A CWF dimension has a name and a length, but unlike netCDF, CWF dimension names are restricted. There are only two allowed dimensions, `rows` and `columns`, and both must be defined before defining a variable.

Dimensions are of type `size_t` rather than `int` to make it possible to access all the data in a CWF dataset on platforms that only support a 16-bit `int` data type, for example MSDOS.

A CWF dimension in an open dataset is referred to by a small non-negative integer, called a *dimension ID*. The library supports the following dimension operations:

- Create a dimension, given its name and length.
- Get a dimension ID from its name.
- Get a dimension's name and length from its ID.

3.1 Create a Dimension: `cw_def_dim`

The function `cw_def_dim` creates a new dimension in an open CWF dataset in define mode. It returns the dimension ID (as an argument) given the dataset ID, and dimension name and length.

Usage

```
int cw_def_dim (int cwid, const char *name, size_t len, int *dimidp);
```

`cwid`	CWF ID, from a previous call to `cw_create`.
`name`	Dimension name. Must be `rows` or `columns`.
`len`	Length of the dimension.
`dimidp`	Pointer to location for returned dimension ID.

Errors

`cw_def_dim` returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The CWF dataset is not in define mode.
- The dimension name is invalid.
- The dimension is already defined.
- The specified length is not greater than zero.

Example

The following example uses `cw_def_dim` to define `rows` and `columns` in a new CWF dataset:

```
#include <cwf.h>
     ...
int status, cwid, rowsid, colsid;
     ...
status = cw_create ("w9813021.lc4", CW_NOCLOBBER, &cwid);
if (status != CW_NOERR) handle_error (status);
     ...
status = cw_def_dim (cwid, "rows", 512, &rowsid);
if (status != CW_NOERR) handle_error (status);
status = cw_def_dim (cwid, "columns", 512, &colsid);
if (status != CW_NOERR) handle_error (status);
```

3.2 Get a Dimension ID from Its Name: `cw_inq_dimid`

The function `cw_inq_dimid` returns (as an argument) a dimension ID, given its name.

Usage

`int cw_inq_dimid (int cwid, const char *name, int *dimidp);`

`cwid`	CWF ID, from a previous call to `cw_open` or `cw_create`.
`name`	Dimension name. Must be `rows` or `columns`.
`dimidp`	Pointer to location for returned dimension ID.

Errors

`cw_inq_dimid` returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The dimension name is invalid.

Example

The following example uses `cw_inq_dimid` to get the dimension ID for `rows`:

```
#include <cwf.h>
     ...
int status, cwid, rowsid;
     ...
status = cw_open ("w9813021.lc4", CW_NOWRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
     ...
status = cw_inq_dimid (cwid, "rows", &rowsid);
if (status != CW_NOERR) handle_error (status);
```

3.3 Inquire about a Dimension: cw_inq_dim

The function cw_inq_dim returns the name and length of a CWF dimension.

Usage

```
int cw_inq_dim (int cwid, int dimid, char *name, size_t *lengthp);
```

cwid	CWF ID, from a previous call to cw_open or cw_create.
dimid	Dimension ID, from a previous call to cw_inq_dimid or cw_def_dim.
name	Returned dimension name. The caller must allocate space for the returned name, up to a maximum of CW_MAX_NAME characters. If name is NULL, the name is not returned.
lengthp	Pointer to location for returned length of dimension. If NULL, the length is not returned.

Errors

cw_inq_dim returns the value CW_NOERR if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The dimension ID is invalid.

Example

The following example uses cw_inq_dim to get the dimension length for rows:

```
#include <cwf.h>
     ...
int status, cwid, rowsid;
size_t rows;
     ...
status = cw_open ("w9813021.lc4", CW_NOWRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
     ...
status = cw_inq_dimid (cwid, "rows", &rowsid);
if (status != CW_NOERR) handle_error (status);
status = cw_inq_dim (cwid, rowsid, NULL, &rows);
if (status != CW_NOERR) handle_error (status);
```

4 VARIABLES

Variables in a CWF dataset are defined when the dataset is created, while in define mode. A variable has a name, type, shape, and data values. A small non-negative integer called a *variable ID* identifies the variable within an open CWF dataset. A variable ID of 0 is assigned to the first variable created, 1 to the next, and so on. A variable can also have associated attributes which identify various properties of the variable such as starting and ending latitude, satellite ID, and projection type (see Section 5 "Attributes").

Operations supported on variables are:

- Create a variable, given its name, data type, and shape.
- Get a variable ID from its name.
- Get a variable's name, data type, shape, and number of attributes from its ID.
- Put an array of values into a variable, given variable ID, corner indices, edge lengths, and a block of values.
- Get an array of values from a variable, given variable ID, corner indices, and edge lengths.

4.1 CWF Variable Limitations

Although in netCDF, a variable can have almost any name, type, and number of dimensions, variables in a CWF dataset are limited. All variables must have exactly two dimensions, `rows` and `columns`, and only certain variable names and types are allowed. The following table lists the supported variable names, units, and the corresponding type constants for defining the variables' external data types in the C interface:

Variable name	**Units**	**C API Mnemonic**
`avhrr_ch1`	albedo × 100 %	`CW_FLOAT`
`avhrr_ch2`	albedo × 100 %	`CW_FLOAT`
`avhrr_ch3`	°C	`CW_FLOAT`
`avhrr_ch4`	°C	`CW_FLOAT`
`avhrr_ch5`	°C	`CW_FLOAT`
`mcsst`	°C	`CW_FLOAT`
`scan_angle`	degrees	`CW_FLOAT`
`sat_zenith`	degrees	`CW_FLOAT`
`solar_zenith`	degrees	`CW_FLOAT`
`rel_azimuth`	degrees	`CW_FLOAT`
`scan_time`	decimal hours	`CW_FLOAT`
`mcsst_split`	°C	`CW_FLOAT`
`mcsst_dual`	°C	`CW_FLOAT`
`mcsst_triple`	°C	`CW_FLOAT`

(table continued ...)

Variable name	Units	C API Mnemonic
cpsst_split	°C	CW_FLOAT
cpsst_dual	°C	CW_FLOAT
cpsst_triple	°C	CW_FLOAT
nlsst_split	°C	CW_FLOAT
nlsst_dual	°C	CW_FLOAT
nlsst_triple	°C	CW_FLOAT
ocean_reflect	albedo × 100 %	CW_FLOAT
turbidity	albedo × 100 %	CW_FLOAT
cloud	—	CW_BYTE
graphics	—	CW_BYTE

Note that although variable names and types are limited, the sizes of the variable dimensions are not. The number of rows and columns can be well beyond the 512 × 512 image that is the norm for most CoastWatch datasets.

Another general limitation of CWF is that there can only be one variable per dataset. This is consistent with how CoastWatch format files have always been distributed – with one image per file. The only exception to this rule is for files that contain graphics planes. Any of the variables with units in °C or albedo are encoded in such a way as to allow for a second variable, called `graphics`, to be included in the dataset as well. The `graphics` variable can accommodate up to four graphics planes by setting the various bits in the graphics byte value. The four most significant bits from the byte are used for encoding the four graphics planes as follows:

bit #	7	6	5	4	3	2	1	0
bit value	128	64	32	16	8	4	2	1
graphics plane #	×	×	×	×	4	3	2	1

For example, a byte value of 1 would represent a graphics bit set for plane 1. Alternately, a byte value of 9 would mean graphics set for planes 1 and 4 and so on.

Although the `graphics` variable can only accommodate four bits of the full byte, it should be noted that no such restriction is placed on the byte values for the `cloud` variable. The full 8 bits are preserved.

4.2 Create a Variable: `cw_def_var`

The function `cw_def_var` adds a new variable to an open CWF dataset in define mode. It returns (as an argument) a variable ID, given the CWF ID, the variable name, the variable type, the number of dimensions, and a list of the dimension IDs.

Usage

```
int cw_def_var (int cwid, const char *name, cw_type xtype, int ndims,
                const int dimids[], int *varidp);
```

`cwid`	CWF ID, from a previous call to `cw_create`.
`name`	Variable name. Must be one of the names previously mentioned.
`xtype`	One of the set or predefined CWF external data types. The type of this parameter, `cw_type`, is defined in the CWF header file. It must correspond to the variable name listed in the variable name/type table, either `CW_FLOAT` or `CW_BYTE`.
`ndims`	Number of dimensions for the variable. Must be 2, or the predefined constant `CW_MAX_VAR_DIMS`.
`dimids`	Vector of `ndims` dimension IDs corresponding to the variable dimensions. The vector must contain the dimension ID for `rows` in `dimids[0]` and `columns` in `dimids[1]`.
`varidp`	Pointer to location for the returned variable ID.

Errors

`cw_var_def` returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The CWF dataset is not in define mode.
- The variable is already defined.
- The variable name is invalid.
- The specified type is invalid – either it does not exist or does not match the variable name.
- The number of dimensions is not 2.
- The dimension IDs in the list of dimensions do not match the dimension IDs for `rows` and `columns`.
- The variable name is `graphics`, but no main data variable is defined which allows for encoding graphics.

Example

The following example uses `cw_var_def` to create a variable called `avhrr_ch4` of type float with two dimensions, `rows` and `columns` in a new CWF dataset named `w9813021.lc4`:

```
#include <cwf.h>
    ...
int status;           /* error status */
int cwid;             /* CWF ID */
int rowsid, colsid;   /* dimension IDs */
int varid;            /* variable ID */
int dimids[2];        /* variable shape */
    ...
status = cw_create ("w9813021.lc4", CW_NOCLOBBER, &cwid);
if (status != CW_NOERR) handle_error (status);
```

```
     ...
                       /* define dimensions */
status = cw_def_dim (cwid, "rows", 512, &rowsid);
if (status != CW_NOERR) handle_error (status);
status = cw_def_dim (cwid, "columns", 512, &colsid);
if (status != CW_NOERR) handle_error (status);
     ...
                       /* define variable */
dimids[0] = rowsid;
dimids[1] = colsid;
status = cw_def_var (cwid, "avhrr_ch4", CW_FLOAT, 2, dimids, &varid);
if (status != CW_NOERR) handle_error (status);
```

4.3 Get a Variable ID from Its Name: `cw_inq_varid`

The function `cw_inq_varid` returns the ID of a CWF variable, given its name.

Usage

```
int cw_inq_varid (int cwid, const char *name, int *varidp);
```

`cwid`	CWF ID, from a previous call to `cw_open` or `cw_create`.
`name`	Variable name for which ID is desired.
`varidp`	Pointer to location for the returned variable ID.

Errors

`cw_inq_varid` returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The specified variable name is not a valid name for a variable in the specified CWF dataset.

Example

The following example uses `cw_inq_varid` to get the variable ID for variable `avhrr_ch4` in dataset `w9813021.lc4`:

```
#include <cwf.h>
     ...
int status, cwid, varid;
     ...
status = cw_open ("w9813021.lc4", CW_NOWRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
     ...
status = cw_inq_varid (cwid, "avhrr_ch4", &varid);
if (status != CW_NOERR) handle_error (status);
```

4.4 Get Information about a Variable from Its ID: cw_inq_var

The function cw_inq_var returns the following information about a variable, given its ID: name, external type, number of dimensions, dimension IDs, and number of attributes.

Usage

```
int cw_inq_var (int cwid, int varid, char *name, cw_type *xtypep,
                int *ndimsp, int dimids[], int *nattsp);
```

cwid	CWF ID, from a previous call to cw_open or cw_create.
varid	Variable ID.
name	Returned variable name. The caller must allocate space for the name, whose maximum possible length is given by CW_MAX_NAME. If this parameter is given as NULL, no name will be returned so no variable to hold the name needs to be declared.
xtypep	Pointer to location for returned variable type, one of the set of predefined CWF external data types. The valid CWF external data types for variables are CW_FLOAT and CW_BYTE. If this parameter is given as NULL, no value will be returned.
ndimsp	Pointer to location for returned number of dimensions. For CWF variables, this will always be 2. If this parameter is given as NULL, no value will be returned.
dimids	Returned vector of *ndimsp dimension IDs corresponding to the variable dimensions. The caller must allocate enough space for a vector of at least *ndimsp integers to be returned. If this parameter is given as NULL, no value will be returned.
nattsp	Pointer to location for returned number of variable attributes assigned to this variable. If this parameter is given as NULL, no value will be returned.

Errors

cw_inq_var returns the value CW_NOERR if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The variable ID is invalid for the specified CWF dataset.

Example

The following example uses cw_inq_var to get information for the variable avhrr_ch4 in dataset w9813021.lc4:

```
#include <cwf.h>
    ...
int status;           /* error status */
int cwid;             /* CWF ID */
```

```
int varid;              /* variable ID */
cw_type xtype;          /* variable type */
int ndims;              /* number of dimensions */
int dimids[2];          /* variable shape */
int natts;              /* number of attributes */
    ...
status = cw_open ("w9813021.lc4", CW_NOWRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_inq_varid (cwid, "avhrr_ch4", &varid);
if (status != CW_NOERR) handle_error (status);
    ...
                        /* use NULL for the name, since we already know it */
status = cw_inq_var (cwid, varid, NULL, &xtype, &ndims, dimids, &natts);
if (status != CW_NOERR) handle_error (status);
```

4.5 Read or Write an Array of Values: `cw_get_vara_`*type* and `cw_put_vara_`*type*

The `cw_get_vara_`*type* and `cw_put_vara_`*type* families of functions read and write values to a variable in an open CWF dataset. The part of the CWF variable to write is specified by giving a corner and a vector of edge lengths that refer to an array section of the CWF variable. The values to be read or written are associated with the variable by assuming that the the `columns` dimension varies fastest. The dataset must be in data mode.

Usage

```
int cw_get_vara_float (int cwid, int varid, const size_t start[],
                       const size_t count[], float *fp);

int cw_get_vara_uchar (int cwid, int varid, const size_t start[],
                       const size_t count[], unsigned char *ucp);

int cw_put_vara_float (int cwid, int varid, const size_t start[],
                       const size_t count[], const float *fp);

int cw_put_vara_uchar (int cwid, int varid, const size_t start[],
                       const size_t count[], const unsigned char *ucp);
```

`cwid`	CWF ID, from a previous call to `cw_open` or `cw_create`.
`varid`	Variable ID.
`start`	A vector of `size_t` integers specifying the index of the variable where the first of the data values will be read or written. The indices are relative to 0, so for example the first data value would have index `(0, 0)`.
`count`	A vector of `size_t` integers specifying the edge lengths along each dimension of the block of data values to be read or written. To read or write a single data value, for example, specify `count` as `(1, 1)`.
`fp, ucp`	Pointer to a block of data values to be read or written. The data is ordered with the `columns` dimension varying fastest. If the type of data values differs from the the CWF variable type, type conversion will occur (see Section 1.4.1 "CWF External Data Types").

Errors

The variable read and write functions return the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The variable ID is invalid for the specified CWF dataset.
- The specified corner indices are out of range of the variable.
- The specified edge lengths added to the corner indices reference data out of range of the variable.
- The type of data conversion required to store the values is not supported.
- The CWF dataset is in define mode rather than data mode.

Example

The following example uses `cw_get_vara_float` and `cw_put_vara_float` to read `avhrr_ch4` data from `w9813021.lc4`, add 0.5 to each value, and rewrite the values back to the dataset.

```
#include <cwf.h>
    ...
#define ROWS          512
#define COLS          512
    ...
int status;                              /* error status */
int cwid;                                /* CWF ID */
int varid;                               /* variable ID */
static size_t start[] = {0, 0};          /* start vector */
static size_t count[] = {ROWS, COLS};    /* count vector */
float ch4[ROWS*COLS];                    /* float data array */
int i;
```

```
    ...
status = cw_open ("w9813021.lc4", CW_WRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_inq_varid (cwid, "avhrr_ch4", &varid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_get_vara_float (cwid, varid, start, count, ch4);
if (status != CW_NOERR) handle_error (status);
for (i = 0; i < ROWS*COLS; i++)
  ch4[i] += 0.5;
status = cw_put_vara_float (cwid, varid, start, count, ch4);
if (status != CW_NOERR) handle_error (status);
```

4.6 Invalid Data Values

Similar to the netCDF concept of a "fill value", there exists a value for variables of type `CW_FLOAT` which is used to represent invalid data. The C constant `CW_BADVAL` is used when a variable's data value is unknown or invalid, ie: the data was never collected at that point or for some reason the value should not be used in calculations. When writing values to a variable, the invalid points can be flagged by either setting them to `CW_BADVAL` in the `float` array, or in a new dataset, by not writing to that block of values at all. In a new dataset all values are initialized to the bad value before any writing can occur. When the CWF library reads a variable that has invalid data points, those points are automatically flagged with `CW_BADVAL` in the `float` array.

The following example could be used to count the number of invalid data values for the variable `avhrr_ch4` in dataset `w9813021.lc4`:

```
#include <stdio.h>
#include <cwf.h>
    ...
#define ROWS          512
#define COLS          512
    ...
int status;                          /* error status */
int cwid;                            /* CWF ID */
int varid;                           /* variable ID */
static size_t start[] = {0, 0};      /* start vector */
static size_t count[] = {ROWS, COLS}; /* count vector */
float ch4[ROWS*COLS];                /* float data array */
int i;
long bad;
    ...
status = cw_open ("w9813021.lc4", CW_WRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_inq_varid (cwid, "avhrr_ch4", &varid);
```

```
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_get_vara_float (cwid, varid, start, count, ch4);
if (status != CW_NOERR) handle_error (status);
for (i = 0, bad = 0; i < ROWS*COLS; i++)
  if (ch4[i] == CW_BADVAL)
    bad++;
printf ("%ld\n", bad);
```

5 ATTRIBUTES

Attributes may be associated with a CWF variable to specify various properties such as the starting and ending latitude, satellite ID, and projection type. Attributes for a CWF dataset are defined when the dataset is created, while the dataset is in define mode, and can be changed later when the dataset is in data mode. A CWF attribute has a variable to which it is assigned, a name, a type, a length, and a sequence of one or more values. An attribute can be referred to either by its variable ID and name, or its variable ID and *attribute ID*. An attribute ID is a small non-negative integer which identifies the attribute within an open CWF dataset.

Operations supported on attributes are:

- Create an attribute, given its variable ID, name, data type, length, and value.
- Get an attribute's data type and length from its variable ID and name.
- Get attribute's value from its variable ID and name.
- Get name of attribute from its attribute ID.

5.1 CWF Attribute Limitations

Just as variables in CWF datasets have limitations, so do attributes. Only certain attribute/type combinations are allowed, and some attributes are limited to having only certain values. All numerical attributes (**float**, **short**) are limited to containing only one value. The `graphics` variable, if it exists, is not allowed to have any attributes.

Unlike netCDF, the CWF interface supports the idea of "read-only" attributes. These are attributes which may be of informational use to the user, but are set internally for use by the interface routines when the file is created.

The following table summarizes the supported attribute names, units, defined external data type constants in C, and restrictions on values (if any):

Attribute name	**Units**	**C API Mnemonic**	**Values**

`satellite_id`	—	`CW_CHAR`	`noaa-6`
			`noaa-7`
			`noaa-8`
			`noaa-9`
			`noaa-10`
			`noaa-11`
			`noaa-12`
			`noaa-14`
			`noaa-15`
			`noaa-16`
			`noaa-17`

`satellite_type`	—	`CW_CHAR`	`morning` `afternoon`

(table continued ...)

Attribute name	Units	C API Mnemonic	Values
data_set_type	—	CW_CHAR	lac gac hrpt
projection_type	—	CW_CHAR	unmapped mercator polar linear
start_latitude	degrees	CW_FLOAT	—
end_latitude	degrees	CW_FLOAT	—
start_longitude	degrees	CW_FLOAT	—
end_longitude	degrees	CW_FLOAT	—
resolution	sampling interval *or* km/pixel *or* degrees/pixel	CW_FLOAT	—
polar_grid_size	—	CW_SHORT	—
polar_grid_points	—	CW_SHORT	—
polar_hemisphere	—	CW_SHORT	—
polar_prime_longitude	degrees	CW_SHORT	—
grid_ioffset	—	CW_SHORT	—
grid_joffset	—	CW_SHORT	—
composite_type	—	CW_CHAR	none nadir average latest warmest coldest
calibration_type†	—	CW_CHAR	raw albedo_temperature
fill_type	—	CW_CHAR	none average adjacent
channel_number†	—	CW_CHAR	avhrr_ch1 avhrr_ch2 avhrr_ch3 avhrr_ch4 avhrr_ch5 mcsst scan_angle sat_zenith solar_zenith rel_azimuth scan_time mcsst_split mcsst_dual

(table continued ...)

Attribute name	Units	C API Mnemonic	Values
			mcsst_triple cpsst_split cpsst_dual cpsst_triple nlsst_split nlsst_dual nlsst_triple ocean_reflect turbidity cloud
data_id†	—	CW_CHAR	visible infrared ancillary cloud
sun_normalization	—	CW_CHAR	yes no
limb_correction	—	CW_CHAR	yes no
nonlinearity_correction	—	CW_CHAR	yes no
orbits_processed	—	CW_SHORT	—
channels_produced†	—	CW_SHORT	—
channel_pixel_size†	—	CW_SHORT	—
channel_start_block	—	CW_SHORT	—
channel_end_block	—	CW_SHORT	—
ancillaries_produced†	—	CW_SHORT	—
ancillary_pixel_size†	—	CW_SHORT	—
ancillary_start_block	—	CW_SHORT	—
ancillary_end_block	—	CW_SHORT	—
image_block_size	—	CW_SHORT	—
compression_type†	—	CW_CHAR	none 1b
percent_non_zero	%	CW_SHORT	—
horizontal_shift	—	CW_SHORT	—
vertical_shift	—	CW_SHORT	—
horizontal_skew	—	CW_SHORT	—
vertical_skew	—	CW_SHORT	—
orbit_type	—	CW_CHAR	ascending descending both
orbit_time	—	CW_CHAR	day night
start_row	—	CW_SHORT	—
start_column	—	CW_SHORT	—

(table continued ...)

Attribute name	Units	C API Mnemonic	Values
end_row	—	CW_SHORT	—
end_column	—	CW_SHORT	—
orbit_start_year	—	CW_SHORT	—
orbit_start_day	—	CW_SHORT	—
orbit_start_month_day	month×100 + day	CW_SHORT	—
orbit_start_hour_minute	hour×100 + minute	CW_SHORT	—
orbit_start_second	—	CW_SHORT	—
orbit_start_millisecond	—	CW_SHORT	—
orbit_end_year	—	CW_SHORT	—
orbit_end_day	—	CW_SHORT	—
orbit_end_month_day	month×100 + day	CW_SHORT	—
orbit_end_hour_minute	hour×100 + minute	CW_SHORT	—
orbit_end_second	—	CW_SHORT	—
orbit_end_millisecond	—	CW_SHORT	—

† Attribute is read-only.

5.2 Read or Write an Attribute: `cw_get_att_`*type* and `cw_put_att_`*type*

The `cw_get_att_`*type* and `cw_put_att_`*type* families of functions read and write attributes for an open CWF dataset. Attributes are defined for a new CWF dataset using `cw_put_att_`*type* while in define mode, and can be re-written while in data mode.

Usage

```
int cw_get_att_text (int cwid, int varid, const char *name, char *tp);

int cw_get_att_short (int cwid, int varid, const char *name, short *sp);

int cw_get_att_float (int cwid, int varid, const char *name, float *fp);

int cw_put_att_text (int cwid, int varid, const char *name,
                     size_t len, const char *tp);

int cw_put_att_short (int cwid, int varid, const char *name,
                      cw_type xtype, size_t len, const short *sp);

int cw_put_att_float (int cwid, int varid, const char *name,
                      cw_type xtype, size_t len, const float *fp);
```

`cwid`	CWF ID, from a previous call to `cw_open` or `cw_create`.
`varid`	Variable ID of the attribute's variable.
`name`	Attribute name. Must be one of the attribute names previously mentioned.
`xtype`	One of the set or predefined CWF external data types. The type of this parameter, `cw_type`, is defined in the CWF header file. It must correspond to the attribute name listed in the attribute name/type table, either `CW_FLOAT`, `CW_CHAR`, or `CW_SHORT`.
`len`	Number of values provided for the attribute. For attributes of type `CW_FLOAT` and `CW_SHORT`, `len` must be 1. For `CW_CHAR`, `len` must be the length of the character string.
`tp, sp, fp`	Pointer to one or more values. If the type of values differs from the CWF attribute type specified as `xtype`, type conversion will occur (see Section 1.4.1 "CWF External Data Types").

Errors

The attribute read and write functions return the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The variable ID is invalid for the specified CWF dataset.
- The specified type is invalid – either it does not exist or does not match the attribute name.
- The specified length is inappropriate for the attribute type.
- The type of data conversion required to store the values is not supported.
- The attribute is read-only, and a put operation was attempted.

Example

The following example uses `cw_get_att_text` and `cw_put_att_text` to switch the `orbit_time` attribute for `avhrr_ch4` in `w9813021.1c4`.

```
#include <cwf.h>
    ...
int status;                          /* error status */
int cwid;                            /* CWF ID */
int varid;                           /* variable ID */
char otime[10];                      /* attribute value */
    ...
status = cw_open ("w9813021.1c4", CW_WRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_inq_varid (cwid, "avhrr_ch4", &varid);
```

```
 if (status != CW_NOERR) handle_error (status);
     ...
 status = cw_get_att_text (cwid, varid, "orbit_time", otime);
 if (status != CW_NOERR) handle_error (status);
 if (strcmp (otime, "day") == 0)
   strcpy (otime, "night");
 else
   strcpy (otime, "day");
 status = cw_put_att_text (cwid, varid, "orbit_time", strlen (otime), otime);
 if (status != CW_NOERR) handle_error (status);
```

5.3 Get Information about an Attribute: cw_inq_att Family

The attribute inquiry family of functions returns the following information about an attribute, given its name: external type, length, and attribute ID. The one exception is cw_inq_attname which returns the attribute name given its ID. This function is useful in generic applications that need to get the names of all the attributes associated with a variable.

Usage

```
int cw_inq_attname (int cwid, int varid, int attid, char *name);

int cw_inq_att (int cwid, int varid, const char *name, cw_type *xtypep,
                size_t *lenp);

int cw_inq_attid (int cwid, int varid, const char *name, int *attidp);
```

cwid	CWF ID, from a previous call to cw_open or cw_create.
varid	Variable ID of the attribute's variable.
name	Attribute name. For cw_inq_attname, this is a pointer to the location for the returned attribute name. The caller must allocate space for the name, whose maximum possible length is given by CW_MAX_NAME.
xtypep	Pointer to location for returned attribute type, one of the set of predefined CWF external data types. The valid CWF external data types for attributes are CW_FLOAT, CW_SHORT, and CW_CHAR. If this parameter is given as NULL, no type will be returned so no variable to hold the type needs to be declared.
lenp	Pointer to location for returned number of values currently stored in the attribute. If this parameter is given as NULL, no value will be returned.
attid	For cw_inq_attname, attribute ID. The attributes for each variable are numbered from 0 (the first attribute) to natts-1, where natts is the number of attributes for the variable, as returned from a call to cw_inq_var.
attidp	For cw_inq_attid, pointer to location for returned attribute ID that specifies which attribute this is for this variable.

Errors

Each function returns the value `CW_NOERR` if no errors occurred. Possible causes of errors include:

- The specified CWF ID does not refer to an open CWF dataset.
- The variable ID is invalid for the specified CWF dataset.
- The attribute does not exist.
- For `cw_inq_attname`, the specified attribute ID is not in the range 0 to `natts-1`.

Example

The following example uses `cw_inq_attname` to get the attribute names for variable `avhrr_ch4` in dataset `w9813021.lc4`:

```
#include <stdio.h>
#include <cwf.h>
    ...
int status;                         /* error status */
int cwid;                           /* CWF ID */
int varid;                          /* variable ID */
int natts;                          /* number of attributes */
char attname[CW_MAX_NAME];          /* attribute name */
int i;
    ...
status = cw_open ("w9813021.lc4", CW_NOWRITE, &cwid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_inq_varid (cwid, "avhrr_ch4", &varid);
if (status != CW_NOERR) handle_error (status);
    ...
status = cw_inq_var (cwid, varid, NULL, NULL, NULL, NULL, &natts);
if (status != CW_NOERR) handle_error (status);
    ...
for (i = 0; i < natts; i++) {
  status = cw_inq_attname (cwid, varid, i, attname);
  if (status != CW_NOERR) handle_error (status);
  printf ("%s\n", attname);
} /* for */
```

6 UTILITIES

This section describes a number of utility routines written to use the Version 2 CWF library. The utilities allow CoastWatch users to:

- convert CoastWatch data to other formats
- query the data dimensions, attributes, and values
- perform simple operations: navigation, statistics, cloud masking, etc.

The utilities are written for use in a command-line oriented operating system. As such, they can be used on any machine running Unix, DOS, or under the Microsoft Windows DOS shell. Most of the utilities print to standard output, the screen by default. Output can be redirected to a file in any of the above systems by using the `>` character (see examples below). DOS and DOS shell users are cautioned not to use output redirection with binary output - rather, use the program's output file option to get correct results.

6.1 Export

6.1.1 Export to ArcView, ARC/INFO: `cwftoarc`

The `cwftoarc` utility converts CWF data to either an ASCII or binary GRID file for use in the ArcView or ARC/INFO GIS packages from Environmental Systems Research Institute, Inc. (ESRI). The file can be used in ArcView with the Spatial Analyst extension, or in ARC/INFO with the ASCIIGRID or FLOATGRID commands, depending on the file type. Note that in binary mode, *two files* are actually created, a `.flt` file to hold the binary data, and a `.hdr` file to specify the geographic location. Both files are required by ArcView or ARC/INFO to import the data.

To help Arc users, `cwftoarc` also outputs a projection specifications note when performing the conversion. The projection specifications *must be copied exactly* in order to correctly overlay other GIS data. In ArcView, the projection is set using the *View|Properties|Projection* dialog box. In ARC/INFO, experienced users should be able to use the projection specifications with the PROJECT command.

Usage

```
cwftoarc [-p fdig] [-d dfdig] [-b bname] input
```

`-p fdig`	Changes the default precision (number of significant digits) for floating-point values (6).
`-d ddig`	Changes the default precision (number of significant digits) for double precision floating-point values (10).
`-b bname`	Changes the output to binary mode, creating <bname>.flt and <bname>.hdr. Default is ASCII mode to standard output.
`input`	The CoastWatch dataset to read.

Example

The following command converts the file `w9823910.js7` for import into ArcView, making sure to output all data values with 4 significant digits rather than the default 6 to reduce the file size:

```
cwftoarc -p 4 w9823910.js7 > w9823910.js7.asc
```

The following command converts `w9823910.js7`, this time in binary mode:

```
cwftoarc -b w9823910 w9823910.js7
```

6.1.2 Export to XYZ ASCII: `cwftoasc`

The `cwftoasc` utility converts CWF data into an ASCII file of (x, y, z) triplets, ie: longitude, latitude, value. The file can subsequently be used in analysis and plotting programs such as the Generic Mapping Tools (GMT)[3]. Note that "bad" or "missing" data values are flagged with the value -999.

Usage

`cwftoasc [-p fdig] [-d ddig] [-r] input`

`-p fdig`	Changes the default precision (number of significant digits) for floating-point values (6).
`-d ddig`	Changes the default precision (number of significant digits) for double precision floating-point values (10).
`-r`	Reverses the default coordinate order (lon, lat).
`-t`	Separates data values with TAB rather than SPACE.
`input`	The CoastWatch dataset to read.

Example

The following command converts `w9823910.js7` into a TAB-delimited ASCII file of (x, y, z) values organized as latitude, longitude, value:

```
cwftoasc -r -t w9823910.js7 > w9823910.js7.asc
```

6.1.3 Export to raw binary: `cwftoraw`

The `cwftoraw` utility converts CWF data into a raw binary file in which each data value is represented by one byte. Byte values are derived from the floating-point data values according to the command line scaling specifications. If the scaling causes a data value to fall outside the output byte range, the byte value will be truncated to the closest representable value. For example, if the scaling calls for data values in the range [0, 25.5] to be mapped to byte values in the range [0, 255], the data value 30.0 will be truncated to byte value 255.

[3]See `http://www.soest.hawaii.edu/soest/gmt.html` for more information on GMT.

Graphics planes can also be included by specifying the desired byte values for each plane. Only the first four graphics planes are currently supported.

Note: When running `cwftoraw` under DOS, use the `-o` option to get correct binary output.

Usage

```
cwftoraw [-# gbyte] [-m maxbyte] [-b badbyte] [-o output] [-l min]
[-h max] input
```

`-# gbyte`	Specifies the byte value for graphics plane `#` (`#` = 1,2,3,4).
`-m maxbyte`	Changes the default maximum byte value (255).
`-b badbyte`	Changes the default bad data byte value (0).
`-o output`	Specifies an output file name, default is write to standard output.
`-l min`	Changes the default minimum data value (0.0).
`-h max`	Changes the default maximum data value (25.5).
`input`	The CoastWatch dataset to read.

Example

The following command scales the temperature data in `w9823910.js7` by a factor of 10, and writes out the byte values to w9823910.js7.raw:

```
cwftoraw w9823910.js7 > w9823910.js7.raw
```

As another example, the following command scales the albedo data in `w9823921.jc2` from albedo values in the range [0, 25] to byte values in the range [0, 250], and applies graphics plane 3 as byte value 255:

```
cwftoraw -3 255 -m 250 -h 25.0 w9823921.jc2 > w9823921.jc2.raw
```

6.1.4 Export to netCDF: `cwftonc`

The `cwftonc` utility converts CWF data to network Common Data Form (netCDF)[4]. Unlike some of the other conversion routines, `cwftonc` retains as much information from the CoastWatch file as possible. All supplementary information including the originating satellite, date, time, projection specifications, etc. is available in the netCDF file. Variable data is encoded as 32-bit floating-point or 16-bit integer values and graphics planes as 8-bit bytes, all with dimensions `rows` and `columns`. Supplementary information is encoded as variable attributes, as described in Section 5. A number of additional attributes may also be included in the file:

`missing_value`	The value for missing or invalid data.
`scale_factor`	Integer data scaling factor; multiply integer data by this value to recover the floating-point data.
`add_offset`	Integer data offset; if non-zero, add this offset after applying the scaling factor to recover the floating-point data.

[4]See `http://www.unidata.ucar.edu/packages/netcdf` for more information on netCDF.

Usage

```
cwftonc [-i] input output
```

`-i`	Encode floating-point data as netCDF 16-bit scaled integers, default is 32-bit floating-point.
`input`	The CoastWatch dataset to read.
`output`	The netCDF dataset to write.

Example

The following command converts `w9823921.jc4` to 16-bit integer netCDF data:

```
cwftonc -i w9823921.jc4 w9823921.jc4.nc
```

6.1.5 Export to HDF: `cwftohdf`

The `cwftohdf` utility converts CWF data to Hierarchical Data Format (HDF)[5]. The routine was completely rewritten between version 2.2 and 2.3 of the utilities; enhancements include:

- standard HDF calibration attributes
- optional use of HDF built-in data compression
- storage of multiple data variables (ie: channel 1, channel 4, SST, etc.)
- conversion of projection attributes to USGS National Mapping Division standard
- reduced output file size due to compression and removal of extra data

`cwftohdf` converts CoastWatch data to HDF, but translates metadata from the CoastWatch file into a standard set of HDF attributes outlined in Appendix B.

Usage

```
cwftohdf [-u] [-v] input1 [input2 [input3 [...]]] output
```

`-u`	Write uncompressed HDF data, default is to use compression.
`-v`	Print verbose status messages.
`input1 input2 ...`	The CoastWatch dataset(s) to read.
`output`	The CoastWatch HDF dataset to write.

Example

The following command converts a number of CoastWatch datasets into one CoastWatch HDF file with HDF compression turned on. Note that all files must have the same date, time, and geographic projection:

[5] See `http://hdf.ncsa.uiuc.edu` for more information on HDF.

```
cwftohdf -v *.cwf combined.hdf
```

6.2 Plotting

6.2.1 Plot to GIF: `cwftogif`

The `cwftogif` utility plots CWF data to a GIF file with accompanying color bar and labels. There are many plotting options as detailed in the usage note below, including:

- rendering graphics planes in a specified color
- scaling data between a minimum and maximum value
- applying one of a number of color maps
- labelling the color bar in Fahrenheit for temperature data
- generating a satellite time stamp

The colors for graphics planes and missing values can be individually specified using a red, green, and blue (RGB) byte-valued intensity triplet, ie: (0,0,0) for black, (0,255,0) for green, (150,150,150) for gray, etc.

Note: Running `cwftogif` under DOS requires the use of `-o`.

Usage

```
cwftogif [-# r,g,b | -# dyn] [-c colors] [-b r,g,b] [-l min] [-h max]
[-p fdig] [-m cmap] [-t ticint] [-f] [-i] [-s [-u]] [-o output] input
```

`-# r,g,b`	Specifies the RGB values for graphics plane `#` (`#` = 1,2,3,4).
`-# dyn`	Specifies that graphics plane `#` should be dynamic, ie: light on dark, dark on light.
`-c colors`	Changes the default number of colors in the palette (246).
`-b r,g,b`	Changes the RGB for bad data values (0,0,0).
`-l min`	Changes the default minimum data value: infrared = 4 visible = 0 angle = 0 scan time = 0 cloud = 0
`-h max`	Changes the default maximum data value: infrared = 30 visible = 20 angle = 90 scan time = 24 cloud = 255
`-d dec`	Changes the default number of decimal places for tick values (1).

`-m cmap`	Changes the default color map:
	infrared = `hsl256`
	visible = `black-white`
	angle = `black-white`
	scan time = `black-white`
	cloud = `hsl256`
	Selections include:
	`black-white`
	`hsl256`
	`prism`

`-t ticint`	Changes the default scale tick interval: infrared = 2 visible = 2 angle = 5 scan time = 2 cloud = 16
`-f`	Changes the plot to work in Fahrenheit, default is Celsius.
`-i`	Inverts the default background and foreground colors (black, white).
`-s`	Places a satellite time stamp on the image, default is no stamp.
`-u`	Formats the satellite time stamp in UTC, default is the local system time zone.
`-o output`	Specifies an output file name, default is write to standard output.
`input`	The CoastWatch dataset to read.

Example

The following command plots `w9823921.jc4` to a GIF with tick marks every 1°C, time stamp in local time, and graphics planes 2 and 3 in dynamic mode:

```
cwftogif -t 1 -s -2 dyn -3 dyn w9823921.jc4 > w9823921.jc4.gif
```

As another example, the following command plots `w9824322.vd7` to a GIF with tick marks every 1°C, inversed color scheme (background white, foreground black), time stamp in local time, graphics planes 2 and 3 in black, and bad or missing values in white:

```
cwftogif -t 5 -i -s -2 0,0,0 -3 0,0,0 -b 255,255,255
w9824322.vd7 > w9824322.vd7.gif
```

6.3 Dataset Information

6.3.1 Get a dimension: `cwfdim`

`cwfdim` prints the size of a specified dimension, either `rows` or `columns`.

Usage

`cwfdim [-d dimension] input`

`-d dimension`	The dimension to print. If not specified, a list of valid dimension names is printed.
`input`	The CoastWatch dataset to read.

Example

The following command prints the number of data rows in `w9823921.jc4`:

```
cwfdim -d rows w9823921.jc4
```

6.3.2 Get an attribute: `cwfatt`

`cwfatt` prints the value of a specified attribute.

Usage

`cwfatt [-a attribute] input`

`-a attribute`	The attribute to print. If not specified, a list of valid attribute names is printed.
`input`	The CoastWatch dataset to read.

Example

The following commands print various useful attributes from `w9823921.jc4`:

```
cwfatt -a satellite_id w9823921.jc4
cwfatt -a projection_type w9823921.jc4
cwfatt -a resolution w9823921.jc4
cwfatt -a channel_number w9823921.jc4
```

```
cwfatt -a orbit_start_year w9823921.jc4
```

6.3.3 Get data values: cwfval

The **cwfval** utility extracts data values from a CWF dataset based on user-specified image coordinates or real world coordinates. The utility can handle either one coordinate pair, or an ASCII file of coordinate pairs. Bad or missing data values are flagged with -999. An attempt to access data outside the data boundaries, either by specifying an invalid image coordinate or by specifying a real world coordinate that transforms to an invalid image coordinate, will produce an error message.

Usage

`cwfval [-p fdig] [-w] -c x y | -f cfile input`

`-p fdig`	Changes the default precision (number of significant digits) for floating-point values (6).
`-w`	Specifies that `x` and `y` are in real world coordinates (longitude, latitude) rather than image coordinates (column, row).
`-c x,y`	Specifies the coordinates of the desired data value. In image coordinates (the default): $1 \leq$ `x` $\leq$ `columns` $1 \leq$ `y` $\leq$ `rows` where `rows` and `columns` are the dataset dimensions. In real world coordinates: $-360 <$ `x` < 360 $-90 <$ `y` < 90 and the data value is approximated using the nearest neighbor.
`-f cfile`	Specifies a file of (x, y) coordinate values, one pair per line, separated by a space.
`input`	The CoastWatch dataset to read.

Example

The following command prints the data value from the center point of `w9823921.jc4`:

```
cwfval -c 256,256 w9823921.jc4
```

6.3.4 Calculate statistics: cwfstats

`cwfstats` calculates the following statistics for a CWF dataset:

- total number of data values
- total number of good data values

- mean
- minimum value
- maximum value
- standard deviation

Usage

`cwfstats input`

`input`	The CoastWatch dataset to read.

Example

The following command runs `cwfstats` on `w9823921.jd7`:

```
cwfstats w9823921.jd7
```

6.3.5 Get CoastWatch HDF file information: `hdfinfo`

The `hdfinfo` routine simply prints a table of information about the contents of a CoastWatch HDF file from `cwftohdf`.

Usage

`hdfinfo input`

`input`	The CoastWatch HDF dataset to read.

Example

The following commands convert a number of CWF files to CoastWatch HDF using `cwftohdf`, then print the file information:

```
cwftohdf *.cwf combined.hdf
hdfinfo combined.hdf
```

Output:

```
Contents of file combined.cwf:

Global information:
  satellite       noaa-14
  sensor          avhrr
  pass_date       1999/10/18
  start_time      20:43:01 UTC
  pass_type       day
```

```
  projection      mercator
  et_affine       1099.964975 0 0 -1099.964975
                  -9872185.654 2820420.762
  rows            512
  cols            512
  origin          USDOC/NOAA/NESDIS CoastWatch HDF library version 3.0

Variable information:
  Variable        Type       Units           Scale        Offset       % Good
  avhrr_ch2       INT16      albedo*100%     0.01         0            100
  graphics        UINT8      -               -            -            -
  avhrr_ch4       INT16      celsius         0.01         0            100
  cloud           UINT8      -               -            -            0
  sst             INT16      celsius         0.01         0            100
```

6.4 Dataset Manipulation

6.4.1 Set navigation attributes: cwfnav

The `cwfnav` utility modifies the `horizontal_shift` and `vertical_shift` attributes in a CWF dataset. Subsequent data reads will take the new shift attributes into account by adjusting the data location with respect to the earth location. Setting a negative or positive horizontal shift will result in the data being shifted left or right respectively. Similarly with the vertical shift, negative or positive will shift the data up or down respectively.

By default, the navigational shifts are applied cumulatively, ie: the shifts written to the file are derived by adding the existing shifts to the user-specified shifts. This behaviour *can* be overridden using the `-r` option to reset the existing navigation before applying the new navigation.

Usage

`cwfnav [-h hshift] [-v vshift] [-r] input`

`-h hshift`	Changes the default horizontal shift (0).
`-v vshift`	Changes the default vertical shift (0).
`-r`	Resets the existing shifts before applying new.
`input`	The CoastWatch dataset to navigate.

Example

The following command applies a navigational shift of 2 pixels left and 3 pixels down to the data in `w9823921.jc4`:

```
cwfnav -h -2 -v 3 w9823921.jc4
```

6.4.2 Apply a cloud mask: `cwfcmask`

The `cwfcmask` utility applies a CoastWatch cloud mask file to a CWF dataset, flagging all cloudy data as invalid. By default, all CLAVR[10] cloud tests employed by CoastWatch are used in applying the cloud mask. A more restricted set of cloud tests can be specified by using the appropriate bits listed in the table below. Note that the cloud tests and corresponding bits vary between daytime and nighttime data. The `orbit_time` attribute may be used to determine which set of bits is appropriate for the data.

CLAVR Test	**Bit**	
	Day	**Night**
Reflective Gross Cloud Test (RGCT)	1	-
Reflectance Uniformity Test (RUT)	2	-
Reflectance Ratio Cloud Test (RRCT)	3	-
Channel 3 Albedo Test (C3AT)	4	-
Thermal Uniformity Test (TUT)	5	2
Four Minus Five Test (FMFT)	6	4
Thermal Gross Cloud Test (TGCT)	7	1
Uniform Low Stratus Test (ULST)	8	3
Cirrus Test (CIRT)	-	5

Usage

`cwfcmask [-b n,n,n ...] cloudmask input output`

`-b n,n,n...`	Specifies the cloud mask bits to use for masking, default is all bits. `n` ranges from 1 to 8.
`cloudmask`	The CoastWatch cloud mask file.
`input`	The CoastWatch dataset to mask.
`output`	The CoastWatch dataset to write. The output values will be altered so that cloudy pixels are flagged as invalid data.

Example

The following command applies the nighttime TGCT and TUT tests from the cloud mask in `e9800306.ncm` to `e9800306.ns7`:

```
cwfcmask -b 1,2 e9800306.ncm e9800306.ns7 e9800306.ns7.masked
```

6.4.3 Create data composite: `cwfcomp`

The `cwfcomp` utility creates a "composite" dataset given two or more CWF datasets. The composite is created from the input datasets by performing calculations on a pixel-by-pixel basis. A number of different calculations are available as described by the formulae below, where N is the number of good pixels from the input datasets at the pixel location:

- mean value:

$$x_{mean} = \left(\sum_{i=1}^{N} x_i \right) / N$$

- median value ($x_1..x_N$ sorted by data value):

$$x_{median} = \begin{cases} x_{(N+1)/2} & \text{N odd} \\ (x_{(N/2)} + x_{(N/2)+1})/2 & \text{N even} \end{cases}$$

- first or last value ($x_1..x_N$ sorted by time):

$$\begin{aligned} x_{first} &= x_1 \\ x_{last} &= x_N \end{aligned}$$

- minimum or maximum value ($x_1..x_N$ sorted by data value):

$$\begin{aligned} x_{min} &= x_1 \\ x_{max} &= x_N \end{aligned}$$

In general, `cwfcomp` is used to create composites from a time series of files, each with the same geographic region and variable type. For example, `cwfcomp` could be used to combine ten days of sea-surface-temperature datasets into one in order to obtain a mean SST field for a certain region and eliminate cloud. Note that for simplicity, the CWF dataset created by `cwfcomp` will indicate the same date and time as the latest file in the time series. It is up to the user to keep track of which datasets have been used in the composite.

Usage

```
cwfcomp [-c ctype] [-v] [-g ngood] [-b badval] input1 input2
[input3 [input4 [...]]] output
```

`-c ctype`	Specifies the composite type. Selections include mean (default), median, first, last, min, and max.
`-v`	Print verbose status messages.
`-g ngood`	Minimum number of good pixels required to form an aggregate function, default is 1.
`-b badval`	Specifies a floating-point value to treat as invalid data, default is none.
`input1 input2`	The CoastWatch datasets to read (at least 2).
`output`	The CoastWatch dataset to write.

Example

The following command creates a mean composite from three CoastWatch datasets, writing output to `mean.sd7`:

```
cwfcomp w9901923.sd7 w9902023.sd7 w9902122.sd7 mean.sd7
```

6.4.4 Create and apply a land mask: `cwflmask`

The **cwflmask** utility uses a combination of polygon filling and morphological transformations[9] to create a land/ocean mask from coastline geography data in graphics plane 3 of a standard CWF dataset. The routine begins with a polygon fill of open ocean pixels at a user-specified row and column. If not all open ocean points are connected (ie: a peninsula or other landform comes between) then a *recursive* fill may be used. A recursive fill can lead to unexpected results when the geography plane contains ambiguous coastline data, such as in the case of very low resolution coastlines or non-coastline features, ie: rivers or state boundaries. Problems can occur with non-recursive polygon fills as well. For example when a poor quality coastline database has been used to render the geography plane, a break in the coastline will cause the fill to spill in and incorrectly identify land as ocean.

After polygon filling, an optional morphological dilation of the land mask can be used to mask out near-land ocean data. This may be desirable if the near-land data is contaminated with a land signal.

Upon output, **cwflmask** stores the generated land mask in graphics plane 1 and masks out all land data from the file, leaving only open ocean data points.

Usage

```
cwflmask [-o r,c] [-r] [-e n] input output
```

`-o r,c`	Specifies a 1-relative row and column point for open ocean, default is to assume that the lower-left corner is an ocean point.
`-r`	Attempt to generate a recursive land mask - may not work well for ambiguous and/or multiply connected coastlines.
`-e n`	Optionally extend the land mask out from the coast by n pixels.
`input`	The CoastWatch dataset to mask.
`output`	The CoastWatch dataset to write. The output values will be altered so that land pixels are flagged as invalid data, and graphics plane 1 will contain the land mask.

Example

The following command applies a recursive land mask to the data in `1999_291_2308_n15_cvc_c4.cwf`, starting the polygon fill at row 256, column 256:

```
cwflmask -o 256,256 -r 1999_291_2308_n15_cvc_c4.cwf test.cwf
```

A QUICK REFERENCE

Datasets

`cw_create`	create dataset
`cw_open`	open existing dataset
`cw_enddef`	end define mode
`cw_close`	close dataset
`cw_strerror`	get error message string

Dimensions

`cw_def_dim`	define dimension
`cw_inq_dimid`	inquire dimension ID
`cw_inq_dim`	inquire dimension info

Variables

`cw_def_var`	define variable
`cw_inq_varid`	inquire variable ID
`cw_inq_var`	inquire variable info
`cw_put_vara_float`	write variable data from `float`
`cw_put_vara_uchar`	write variable data from `unsigned char`
`cw_get_vara_float`	read variable data into `float`
`cw_get_vara_uchar`	read variable data into `unsigned char`

Attributes

`cw_inq_attname`	inquire attribute name
`cw_inq_att`	inquire attribute info
`cw_inq_attid`	inquire attribute ID
`cw_put_att_text`	write attribute from `char`
`cw_put_att_short`	write attribute from `short`
`cw_put_att_float`	write attribute from `float`
`cw_get_att_text`	read attribute into `char`
`cw_get_att_short`	read attribute into `short`
`cw_get_att_float`	read attribute into `float`

Utilities

`cwftoarc`	export to ArcView
`cwftoasc`	export to ASCII
`cwftoraw`	export to binary
`cwftonc`	export to netCDF
`cwftohdf`	export to HDF
`cwftogif`	plot to GIF
`cwfdim`	get dimension
`cwfatt`	get attribute

Utilities	*(continued ...)*
`cwfval`	get data
`hdfinfo`	examine CoastWatch HDF
`cwfnav`	set navigation
`cwfcmask`	apply cloud mask
`cwfstats`	calculate statistics
`cwfcomp`	create composite
`cwflmask`	create/apply land mask

B COASTWATCH HDF METADATA SPECIFICATION

CoastWatch HDF files created by cwftohdf via the HCWF library follow a number of conventions for storing CoastWatch satellite data in HDF format:

1) Multiple channels and derived varibles can be stored in one HDF file. A standard CoastWatch product file contains data from one time (ie: satellite pass) and CoastWatch region only.

2) A standard set of global attributes is encoded with the data, describing the time, location, satellite, sensor, etc. from which the data originated.

3) A standard set of variable attributes is encoded with each variable, describing the variable units, scaling factor, etc. as well as any other important information such as the equations and corrections used in data processing.

The following table lists the standard set of global attributes for CoastWatch HDF. Since all map projection calculations in the HCWF library are performed using the General Cartographic Transformation Package (GCTP) from the USGS National Mapping Division, a number of global attributes are dedicated to storing GCTP-related parameters. See the GCTP documentation for details on the values of GCTP parameters.

NAME	TYPE	DESCRIPTION
satellite	CHAR8	Satellite name, eg: noaa-12, noaa-14, noaa-15, goes-8, orbview-2.
sensor	CHAR8	Sensor name, eg: avhrr, seawifs.
pass_date	INT32	Date of satellite pass in days since January 1, 1970.
start_time	FLOAT64	Start time of satellite pass in seconds since 00:00:00 UTC.
pass_type	CHAR8	Satellite pass time: day, night.
projection	CHAR8	Descriptive projection name, eg: mercator, geographic, polar stereographic.
gctp_sys	INT32	GCTP projection system code.
gctp_zone	INT32	GCTP zone for UTM projections.
gctp_parm	FLOAT64	GCTP projection parameters (15).
gctp_datum	INT32	GCTP spheroid code.
et_affine	FLOAT64	Earth transform affine parameters (6) - see below for details.
rows	INT32	Number of data rows.

cols	INT32	Number of data columns.
origin	CHAR8	Original data source, eg: USDOC/NOAA/NESDIS CoastWatch.
history	CHAR8	Newline separated list of utilities and command line parameters used to create the file and perform subsequent processing.

The et_affine attribute is used by the HCWF library to calculate projection (x,y) coordinates from image (i,j). GCTP is then used to calculate (latitude,longitude) from (x,y). Given the six affine transform parameters as follows:

```
  a = et_affine[0]            e = et_affine[4]
  b = et_affine[1]            f = et_affine[5]
  c = et_affine[2]
  d = et_affine[3]
```

the following vector calculation is performed:

```
  |x|    |a b|  |i|       |e|
  | | =  |   |  | |   +   | |         ( or  Y = AX + B )
  |y|    |c d|  |j|       |f|
```

where (x,y), (i,j), and (e,f) are column vectors, (a,b,c,d) is a 2x2 matrix, and:

```
  x = easting
  y = northing
  i = column (1-relative)
  j = row (1-relative)
```

The inverse operation may be performed by inverting the affine transform:

```
  det(A) = ad - bc
  a' = d / det(A)
  b' = -b / det(A)
  c' = -c / det(A)
  d' = a / det(A)
  e' = -(a'e + b'f)
  f' = -(c'e + d'f)
```

so that:

```
  |i|    |a' b'|  |x|       |e'|
```

```
| | = |      |  | |   +   |  |         ( or  X = A'Y + B' )
|j|   |c' d'|  |y|        |f'|
```

The following table shows the standard set of variable attributes for CoastWatch HDF. Some attribute groups are created by HDF SD convenience functions, denoted in brackets (), in order to make data more readable and usable by generic HDF viewing programs. A <var> in the TYPE field indicates that the attribute type is the same as the variable data type.

NAME	TYPE	DESCRIPTION
(SDsetdatastrs)		
long_name	CHAR8	Descriptive variable name, eg: AVHRR channel 4, sea surface temperature.
units	CHAR8	Descriptive units name, eg: celsius, albedo*100%, degrees.
format	CHAR8	FORTRAN-77 notation for data value printing, eg: F7.2.
coordsys	CHAR8	Coordinate system - same as global projection attribute.
(SDsetfillvalue)		
_FillValue	<var>	Value used to fill in for unwritten data.
(SDsetcal)		
scale_factor	FLOAT64	Calibration scale factor.
scale_factor_err	FLOAT64	Calibration scale error.
add_offset	FLOAT64	Calibration offset.
scale_factor	FLOAT64	Calibration offset error.
calibrated_nt	INT32	Code for HDF data type of uncalibrated data.
C_format	CHAR8	C notation for data value printing, eg: %7.2f.
missing_value	<var>	Value used for missing data, same as _FillValue attribute.
solar_corr	CHAR8	For AVHRR channel 1 and 2 data, whether the solar zenith angle correction was performed: yes, no.
limb_corr	CHAR8	For AVHRR channel 4 and 5 data, whether the limb correction was performed: yes, no.
nonlinear_corr	CHAR8	For AVHRR channel 4 and 5 data, whether the nonlinearity correction was performed:

		yes, no.
sst_equation	CHAR8	For sea surface temperature, the SST equation used, eg: nonlinear split-window.
percent_good	INT16	Good pixels / total pixels * 100%.

The calibration attributes are used by the CWFv3 library to read and write channel and ancillary data as follows:

```
float = scale_factor*(int - add_offset)         ( on read )
int = float/scale_factor + add_offset           ( on write )
```

where float and int are the floating-point and integer values respectively. See the HDF User's Guide for more details on data calibration.

Note that not all variable attributes are required for any given CoastWatch variable; for example the calibration attributes are not needed for graphics data since graphics planes are encoded as 8-bit bytes and require no calibration. Also some attributes such as solar_corr, limb_corr, nonlinear_corr, and sst_equation only have meaning with certain variables.

References

[1] *Map Projections.* Environmental Systems Research Institute, Inc., USA, 1994.

[2] *HDF User's Guide, Version 4.1r2.* University of Illinois at Urbana-Champaign, June, 1998.

[3] *HDF Reference Manual, Version 4.1r2.* University of Illinois at Urbana-Champaign, June, 1998.

[4] Kernighan, B. and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall Inc., USA, 1988.

[5] Kidwell, K. B. (Editor) *NOAA POLAR ORBITER DATA USER'S GUIDE.* USDOC/NOAA/NESDIS, January, 1997.

[6] Mulligan, P. (prepared for) *Interface Control Document for NOAA-K, L & M Environmental Image Products.* Contract No. 50-DDNE-9-00018, USDOC/NOAA/NESDIS, August 21, 1992.

[7] Mulligan, P. (prepared for) *User's Manual for the NOAA-K, L & M Environmental Image Products.* Contract No. 52-DDNE-9-00018, USDOC/NOAA/NESDIS, August 25, 1992.

[8] Rew, R., G. Davis, S. Emmerson, and H. Davies. *NetCDF User's Guide for C: An Access Interface for Self-Describing, Portable Data, Version 3.* Unidata Program Center, University Corporation for Atmospheric Research, Boulder, Colorado, 1997.

[9] Simpson, James J. *Image Masking Using Polygon Fills and Morphological Transformations*, REMOTE SENS. ENVIRON. 40:161-183, 1992.

[10] Stowe, L. L. *et al. Global distribution of cloud cover derived from NOAA/AVHRR operational satellite data.* Adv. Space Res. Vol. 11 No. 3 pp. (3)51-(3)54, 1991.