CLIMSAT "SCAN DATA" FORMAT
June 2000

I. Data Format

1. Data stored in this format are stored by scanline swath. The data files begin with 5000 bytes of header information describing the file contents. The header is followed by records (one for each pixel) storing a time, a latitude, a longitude, and a series of values (e.g. brightness temperatures or estimated integrated water vapor) for each of N channels.

The **header data** (first 5000 bytes) at the top of each datafile have the following format:

bytes	start byte	description
char*80	0	name of file
char*20	80	name of satellite
char*20	100	name of sensor
int*2	120	satellite ID
int*2	122	number of data fields per (low-resolution) pixel
int*2	124	number of (low-resolution) pixels per scanline
int*2	126	number of high-resolution data fields per pixel (0 if N/A)
int*2	128	number of high-resolution pixels per scanline (0 if N/A)
int*2	130	missing value (e.g. -9999)
float*4	132 etc.	scaling factor for each field |
float*4	136 etc.	offset value for each field |-- Repeated for N fields
char*40	140 etc.	units for each field |
char*80	180 etc.	description for each field |
N bytes		Remaining bytes are filler, up to the 5000th byte

Note: when extracting data, *scale* and *offset* should be applied as follows:
1. *First*, divide the resulting value by the *scale* factor;
2. *Second*, subtract the *offset* value from the data.

The **observational data** (e.g., brightness temperatures) begin at the 5001st byte of the file, with one record of data for each pixel. Each record has this format:

int*4	integer time in seconds from 1-1-1970
int*2	latitude in decimal degrees, scaled by 100
int*2	longitude in decimal degrees, scaled by 100
int*2	TB data value, Field 1 (scaled, offset, and described as indicated in header)
int*2	TB data value, Field 2 (scaled, offset, and described as indicated in header)
...	

int*2 TB data value, Field N (scaled, offset, and described as indicated in header)

The **end-of-file** is marked by a bogus data record whose integer time is set equal to the missing value (the value is specified in the header).

2. DMSP datasets may store a single resolution of data (e.g. 19 and 37 Ghz) or may store low- and high-resolution data together in a single file (e.g. 19 and 37 Ghz (low-res) with hi-res 85 Ghz). The spatial configuration of data from low- and high-resolution scanlines is illustrated in the following figure:

Figure: Spatial configuration of low- and high-resolution DMSP SSM/I data.

When **both high-resolution and low-resolution data** are stored in the same data file, the format of the TB data is as follows:

Record 0 through Record N store sequential data for **scanline A**, where:

- **even records** record itime, lat, lon, and all coincident ***low-resolution and high-resolution*** data fields
- **odd records** record itime, lat, lon, and all intermediate ***high-resolution*** data fields

Record N+1 through N+N store sequential data for the **scanline B**, where:

- all records record itime, lat, lon, and all ***high-resolution*** data fields

This sequence is continued through the end of the file, alternating records of dual-resolution and single-resolution scanline data in the order that the data were originally scanned.

II. Code for Reading Data (look under /home/climsat/src/grid_data)

CDC has written data processing code in C to assist users in reading and extracting data stored in the above format. An attempt has been made to document this code as clearly as possible. The following functions are available (and an example of their use is provided as **read_dmsp.c**):

1. read_hdr.c

Fills the passed *std_hdr* structure with header data read from the first 5000 bytes of the specified file. Allocates the memory required to store the header data. Returns 0 if successful, <0 if an error.

The *std_hdr* structure is defined as follows:

```
struct std_hdr
 {
  char    filename [80] ;              /* Name of file                          */
  char   satellite [20] ;              /* Name of satellite                     */
  char    sensor[20] ;                 /* Name of sensor                        */
  short   satid ;                      /* Satellite ID                  */
  short   fields ;                /* Number of (low-resolution) fields   */
  short  pixperscan ;                  /* Pixels per (low-resolution) scanline */
  short   hifields ;                   /* Number of high-resolution fields    */
  short   hipixperscan ;               /* Pixels per high-resolution scanline  */
  short   missing_val ;                /* Value for missing data                */
  float   *scale ;                     /* Scaling factor for each field         */
  float   *offset ;                    /* Offset for each field         */
  struct units_text *units ;           /* Units for each field                  */
  struct describe_text *description    /* Description for each field            */
 }

struct describe_text
 {
  char    text[80] ;                   /* Description of data field             */
 }

struct units_text
 {
  char    text[40] ;                   /* Units of data field                   */
 }
```

2. read_scan.c

Fills the passed *std_data* structures for low-resolution, high-resolution scan A, and high-resolution scan B data with the next scanline of data. Memory must be allocated for each required structure before calling this function (see *allocdata* below). Returns 0 if successful, 999 if the end-of-file record is read, <0 if an error.

The *std_data* structure is as follows:

```
struct std_data
 {
  long    *itime ;             /* Seconds since 1-1-1970                */
  float   *lat ;               /* Decimal degrees latitude * 100        */
  float   *lng ;               /* Decimal degrees longitude * 100       */
```

```
  float   **fielddata ;          /* Data.  Dimensions: [fields][pixels]          */
}
```

3. allocdata.c

Allocates the necessary memory for the passed *std_data* structure to hold one scanline of data. The necessary arguments for number of pixels per scan and number of fields can be derived from the filled *std_hdr* structure. Returns 0 if successful, <0 if an error.

III. Executables

1. view_header

Prints out the header information for a given file including the filename, sensor ID, and data fields along with scale, offset, and description. Also prints out the start and end times and the number of scans in the file. This is basically for querying the file to determine its contents. Usage: *view_header filename.*

2. read_swath

Prints out swath data for the specified file and specified swaths including the latitude, longitude, and corresponding data values for each field. Useful for verifying the data contents of a file in swath format. Usage: type *read_swath*, and the program prompts for user-supplied information.

3. convert_time

Converts from time in seconds starting Jan 1 of a user specified baseline year (1970 for the SSM/I and SSM/T2 files) to a readable date and time string.

4. swapscans

Performs a byte-swap on a Climsat scan data formatted file, such that a scan file created on a little-endian machine can be read on a big-endian machine, and vice-versa.

IV. Use of the Data Reading Functions

The functions *read_hdr*, *read_scan*, and *allocdata* should be used to read and write data. Essentially, the steps for using these functions are as follows:

1. Open input datafile.

2. Read the header data into the structure *std_hdr* using the function *read_hdr*.

3. Allocate the memory required to hold one scanline of data in the structure *std_data* using the function *allocdata.* One structure is required to hold single-resolution data; two additional *std_data* structures are required if dual-resolution data are being read: one for high-resolution data from scanline A, and one for high-resolution data from scanline B.

4. Read the first scanline of data using the function *read_scan* beginning at byte 5000.

5. Do whatever needs to be done with the scanline of data loaded into the *std_data* structure(s).

6. Continue calling *read_scan* to read scanlines until *read_scan* returns 999 (end-of-file).

The code called **read_dmsp.c** is an example of how to read and write scanline data in this format.