<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTER_AND_BAND_VALUES</td>
<td>197</td>
</tr>
<tr>
<td>COLLAPSE_DIMENSIONS</td>
<td>203</td>
</tr>
<tr>
<td>FILL_DATA</td>
<td>211</td>
</tr>
<tr>
<td>FILL_DATA_ENVELOPE</td>
<td>219</td>
</tr>
<tr>
<td>FILL_DISCONTINUOUS_DATA</td>
<td>227</td>
</tr>
<tr>
<td>FILL_MODE_DATA</td>
<td>235</td>
</tr>
<tr>
<td>FILL_MODE_INFO</td>
<td>241</td>
</tr>
<tr>
<td>FILL_SENSOR_INFO</td>
<td>245</td>
</tr>
<tr>
<td>FILL_THETA_MATRIX</td>
<td>249</td>
</tr>
<tr>
<td>MODE_UNITS_INDEX</td>
<td>255</td>
</tr>
<tr>
<td>NUMBER_OF_DATA_BINS</td>
<td>259</td>
</tr>
<tr>
<td>NUMBER_OF_PHI_BINS</td>
<td>263</td>
</tr>
<tr>
<td>RETURN_CENTER_AND_BAND_PTRS</td>
<td>267</td>
</tr>
<tr>
<td>RETURN_PHI_PTRS</td>
<td>271</td>
</tr>
<tr>
<td>SET_BIN_INFO</td>
<td>275</td>
</tr>
<tr>
<td>SET_COLLAPSE_INFO</td>
<td>285</td>
</tr>
<tr>
<td>SET_SCAN_INFO</td>
<td>289</td>
</tr>
<tr>
<td>SET_TIME_VALUES</td>
<td>293</td>
</tr>
<tr>
<td>SPIN_DATA</td>
<td>295</td>
</tr>
<tr>
<td>SPIN_DATA_PIXEL</td>
<td>301</td>
</tr>
<tr>
<td>SWEEP_DATA</td>
<td>307</td>
</tr>
<tr>
<td>SWEEP_DISCONTINUOUS_DATA</td>
<td>313</td>
</tr>
<tr>
<td>SWEEP_MODE_DATA</td>
<td>321</td>
</tr>
<tr>
<td>UNITS_INDEX</td>
<td>327</td>
</tr>
<tr>
<td>CREATE_SCF_DATA_STRUCTURE</td>
<td>331</td>
</tr>
<tr>
<td>FREE_SCF_INFO</td>
<td>335</td>
</tr>
<tr>
<td>INIT_SCF</td>
<td>337</td>
</tr>
<tr>
<td>LOAD_SCF</td>
<td>339</td>
</tr>
<tr>
<td>READ_SCF</td>
<td>343</td>
</tr>
<tr>
<td>SCF_OPEN</td>
<td>347</td>
</tr>
<tr>
<td>SCF_OUTPUT_DATA</td>
<td>351</td>
</tr>
<tr>
<td>SCF_POSITION</td>
<td>357</td>
</tr>
<tr>
<td>SCF_SAMPLE_RATE</td>
<td>361</td>
</tr>
<tr>
<td>SCF_TERMINATE_SOURCES</td>
<td>365</td>
</tr>
<tr>
<td>SCF_VERSION_NUMBER</td>
<td>369</td>
</tr>
<tr>
<td>SCF_ALGORITHM_START</td>
<td>371</td>
</tr>
<tr>
<td>SCF_BIN_INFO</td>
<td>375</td>
</tr>
<tr>
<td>SCF_OUTPUT_CENTER_AND_BANDS</td>
<td>383</td>
</tr>
<tr>
<td>SCF_OUTPUT_DATA_INDEX</td>
<td>387</td>
</tr>
<tr>
<td>SCF_OUTPUT_SELECT</td>
<td>391</td>
</tr>
<tr>
<td>SCF_SAMPLE_AVERAGE</td>
<td>395</td>
</tr>
<tr>
<td>SCF_TIME_AVERAGE</td>
<td>399</td>
</tr>
<tr>
<td>SCF_TIME_REFERENCE</td>
<td>405</td>
</tr>
<tr>
<td>LIBBASE_IDFS.H</td>
<td>407</td>
</tr>
<tr>
<td>RET_CODES.H</td>
<td>409</td>
</tr>
</tbody>
</table>
USER_DEFS.H ................................................................. 421
LIBTREC_IDFS.H ............................................................ 425
SCF_CODES.H ................................................................. 427
SCF_DEFS.H ................................................................. 431
SCF_FILE_DEFS.H .......................................................... 433
LIBBASE_SCF.H ............................................................. 435
LIBAVG_SCF.H ............................................................... 437
IDF_DATA ........................................................................ 439
DIRECTION_COS ............................................................ 445
TRANSFORMATION_INFO .................................................... 447
SCF_DATA ........................................................................ 451
TENSOR_DATA ................................................................. 453
## Revision Log

<table>
<thead>
<tr>
<th>Revision</th>
<th>Release Date</th>
<th>Changes to Document</th>
</tr>
</thead>
</table>
| A        | 09/12/2002   | - Manual was re-formatted and modifications were made to bring the manual up-to-date with respect to code.  
- Calling sequence changed for `set_bin_info()`  
- Calling sequence changed for `collapse_dimensions()`  
- Changes were made to update the error codes returned by the SCF modules since tensor manipulation was implemented. |
| B        | 01/07/2003   | - Added the module `fill_data_envelope()` |
| C        | 10/20/2003   | - Calling sequence changed for `set_bin_info()` and `scf_bin_info()` since added new variable spacing format 'A' for actual values (no computations) |
| D        | 08/23/2004   | - Updated SYNOPSIS and EXAMPLES sections for `fields_to_key()`  
- Updated DESCRIPTION and EXAMPLES sections for `init_idfs()` and `init_scf()` to reference `dbInitialize()` and `CfgInit()`  
- Updated ARGUMENTS section for better clarification for `set_bin_info()`  
- Added the module `first_idfs_sensor()`  
- Calling sequence changed for `collapse_dimensions()`  
- Calling sequence changed for `return_phi_ptrs()`  
- Updated ARGUMENTS, DESCRIPTION and EXAMPLES section for `scf_bin_info()`  
- Added the module `scf_terminate_sources()`  
- Updated `ret_codes` header file (1H) |
| E        | 06/16/05     | - Table of Contents added  
- Added another example between example 5 and example 6, so renumbered examples 6 - 8  
- Added 1R modules `create_tensor_data_structure()`, `extract_from_idfs_tensor()`, `read_drec_spin()`, `read_tensor_data()`, `start_of_spin()`, `start_spin_source_status()`, and `valid_tensor_data_structure()`  
- Added 2R modules `spin_data()`, `spin_data_pixel()` |
<p>| F        | 12/15/05     | - <code>convert_to_units()</code>, <code>fill_sensor_info()</code>, <code>fill_mode_info()</code>, <code>mode_units_index()</code>, <code>units_index()</code>, <code>set_bin_info()</code>, and <code>set_scan_info()</code> changed the data type for the <code>tbl_oper</code> argument(s) since table operators were expanded from 2-byte to 4-byte values for the additional data buffer capabilities |</p>
<table>
<thead>
<tr>
<th>Revision</th>
<th>Release Date</th>
<th>Changes to Document</th>
</tr>
</thead>
</table>
| F        | 12/15/05     | • Added error codes -438 through -461 to `ret_codes.h` header file  
• Added definition of `MAX_UNITS_BUFFERS` to `user_def.h` header file, needed for the data buffer capabilities that were added for unit conversion  
• Added write-up for `potential_source_status()` and modified write-up for `idf_data.h` since added capability to return spacecraft potential data from `read_drec()`  
• Updated status codes for `file_open()`, `file_pos()`, `read_drec()` and `read_tensor_data()` for error codes associated with spacecraft potential data. |
| G        | 06/14/06     | • Added / removed error codes from `ret_codes.h` header file and added to / removed from modules that return these error codes. |
| H        | 06/30/06     | • Updated `file_pos()`, `free_experiment_info()` for tensor_data or idf_data usage for data structure parameter  
• Modified `get_data_key()` to make references to the include file `libdb.h` which is needed for the prototype since it was moved out of `libbase_idfs.h`  
• Updated calling sequence for `sweep_data()`, `sweep_mode_data()` and `sweep_discontinuous_data()` and updated DESCRIPTION section for spacecraft potential data  
• Added spacecraft potential as data type for `units_index()`, `convert_to_units()`, and `fill_sensor_info()` and updated DESCRIPTION section  
• Updated DESCRIPTION section for spacecraft potential data for `file_open()`, `file_pos()`, `read_drec()`, `read_drec_spin()`, `reset_experiment_info()`, `fill_data()`, `fill_data_envelope()`, `fill_discontinuous_data()`, `fill_mode_data()`, `spin_data()`, and `spin_data_pix()`  
• Added `SCF_TENSOR_VECTOR_SRC` error code to `scf_output_data()` and `SCF_codes` include file (3H) |
<p>| I        | 08/17/06     | • Added new module <code>override_potential_polynomial()</code> and added new error codes to <code>ret_codes.h</code> header file. |
| J        | 12/18/06     | • Added new modules <code>turn_off_pitch_angle_computations()</code> and <code>turn_on_euler_angle_computations()</code> for speed issues. |</p>
<table>
<thead>
<tr>
<th>Revision</th>
<th>Release Date</th>
<th>Changes to Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>12/18/06</td>
<td>• Updated <code>ret_codes.h</code>, <code>user_defs.h</code>, and <code>idf_data.h</code> header files since added capability to return euler angle data from <code>read_drec()</code>.&lt;br&gt;• Modified <code>file_pos()</code>, <code>file_open()</code>, <code>read_drec()</code>, and <code>read_tensor_data()</code> for error codes associated with euler angle data.&lt;br&gt;• Added modules <code>euler_angle_source_status()</code></td>
</tr>
<tr>
<td>K</td>
<td>09/29/08</td>
<td>• Added clarification to description section for <code>override_potential_polynomial()</code></td>
</tr>
<tr>
<td>L</td>
<td>07/14/09</td>
<td>• Added module <code>create_data_structure()</code>&lt;br&gt;• Modified calling sequence for <code>center_and_band_values()</code>&lt;br&gt;• Updated <code>ret_codes.h</code> – moved all positive status codes to beginning and added <code>CENTER_CONVERSION</code>, <code>READ_SPIN_DATA_GAP</code>, <code>WRONG_DATA_STRUCTURE</code> and <code>CREATE_DSTR_NOT_FOUND</code></td>
</tr>
<tr>
<td>M</td>
<td>09/24/09</td>
<td>• Added new modules for coordinate system transformations:&lt;br&gt;<code>turn_on_celestial_position_computations()</code>, <code>celestial_position_source_status()</code>&lt;br&gt;• Updated <code>ret_codes.h</code> for coordinate system transformations – added&lt;br&gt;<code>TURN_ON_CP_NOT_FOUND</code>, <code>UPDATE_IDF_BAD_CP_DEF</code>, <code>CP_MAIN_DATA_MISSING</code>, <code>CP_STR_MALLOC</code>, <code>CP_DATA_MALLOC</code>, <code>FILE_POS_CP</code>, <code>CP_INFO_IDF_ELE_NOT_FOUND</code>, <code>CP_INFO_IDF_MANY_BYTES</code>, <code>CP_INFO_IDF_TBL_NUM</code>, <code>CP_INFO_IDF_CON_NUM</code>, <code>CP_INFO_IDF_NO_ENTRY</code>, <code>CP_TBL_MALLOC</code>, <code>CP_BAD_SRC</code>, <code>BAD_CP_FORMAT</code>, <code>NO_CP_CONSTANT</code>, <code>RESET_CP_REALLOC</code>, <code>CP_BAD_FRAC</code>, <code>CP_BAD_TIMES</code>, <code>RESET_TINFO_MALLOC</code>&lt;br&gt;• Updated <code>user_defs.h</code> to add coordinate system transformation mneumonics&lt;br&gt;• Removed unused error codes from <code>ret_codes.h</code>, which include –&lt;br&gt;<code>NUM_CAL_MALLOC</code>, <code>OPEN_EX_MALLOC</code>, <code>LOCATE_EX_MALLOC</code>, and <code>VIDF_OPEN_EX_MALLOC</code></td>
</tr>
<tr>
<td>Revision</td>
<td>Release Date</td>
<td>Changes to Document</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>---------------------</td>
</tr>
</tbody>
</table>
| M        | 09/24/09     | • Modified error codes returned by `file_open()`, `file_pos()`, `read_drec()`, and `read_tensor_data()` to reflect codes associated with coordinate system transformation code  
• Modified `idf_data` structure for coordinate system transformation changes and added write-up for `idfs_transformation` structure  
• Updated `tensor_data` structure since out of revision  
• Updated example section in `fields_to_key()` to utilize SDDAS data types  
• Modified `read_drec_spin()` since calling sequence was incorrect  
• Updated error codes returned by `scf_output_data()` and `scf_position()`  
• Removed some error codes returned by `collapse_dimensions()` that are no longer applicable  
• Updated calling sequences for `spin_data()` and `spin_data_pixel()` – changes were needed in case of coordinate system transformation  
• Updated error codes returned by `load_scf()` |
| N        | 08/31/11     | • Updated description section for `turn_off_pitch_angle_computations()`, `turn_on_euler_angle_computations()` and `turn_on_celestial_position_computations()` since must be called BEFORE `file_pos()`, not `read_drec()`  
• Updated description section for `file_pos()` to indicate that data structures have been filled in and can be interrogated upon return and updated error codes  
• Deleted write-ups for `start_spin_source_status()`, `potential_source_status()`, `pitch_angle_source_status()`, `euler_angle_source_status()` and `celestial_position_source_status()` since no longer available  
• Changed name from `extract_from_idfs_tensor()` to `extract_single_element_from_idfs_tensor()`  
• Moved status codes associated with ancillary data from `file_open()` to `file_pos()` since ancillary data is now processed AFTER the main IDFS source has been positioned successfully  
• Updated error codes for `read_tensor_data()` since data quality flags can now be a tensor as well  
• Updated error codes for `collapse_dimensions()` |
<table>
<thead>
<tr>
<th>Revision</th>
<th>Release Date</th>
<th>Changes to Document</th>
</tr>
</thead>
</table>
| N        | 08/31/11     | • Removed `HDR_FMT_TWO`… error codes from list of error codes returned by `read_drec()` since only pertinent to multi-dimensional data  
• Updated `SCF_DEFS.H` in section 3H  
• Modified data type for `cal_len` and `cset_num` elements in `idf_data` structure in section 1S  
• Modified `tensor_data` structure in section 1S |
| O        | 06/04/12     | • Added new modules `destroy_last_idf_data_structure()` and `destroy_last_tensor_data_structure()`  
• Updated error codes for `file_pos()` since now need to make sure address of data structure is valid  
• Updated `ret_codes.h` header file since added these 2 new modules and modified `file_pos()` |
| P        | 12/28/12     | • Updated `ret_codes.h` for background data – added `BKGD_MAIN_DATA_MISSING`, `BKGD_BAD_SRC`, `BAD_BKGD_FORMAT`, `BKGD_TBL_MALLOC`, `BKGD_MALLOC`, `BKGD_DATA_MALLOC`, `BKGD_IDF_DATA_MALLOC`, `BKGD_INFO_IDF_ELE_NOT_FOUND`, `BKGD_INFO_IDF_MANY_BYTES`, `BKGD_INFO_IDF_TBL_NUM`, `BKGD_INFO_IDF_CON_NUM`, `BKGD_INFO_IDF_NO_ENTRY`, `FILE_POS_BKGD`, `BKGD_BAD_TIMES`, `RESET_BKGD_REALLOC`, `BKGD_BAD_FRAC`, `UPDATE_IDF_BAD_BKGD_DEF`, `NO_BKGD_CONSTANT`  
• Modified error codes returned by `file_open()`, `file_pos()`, `read_drec()` and `read_tensor_data()` to reflect codes associated with background data  
• Updated `user_defs.h` and `idf_data.h` header files since added capability to return background data from `read_drec()`  
• Added background as data type for `units_index()`, `convert_to_units()`, and `fill_sensor_info()` and updated `DESCRIPTION` section  
• Updated `DESCRIPTION` section for background data for `file_open()`, `file_pos()`, `read_drec()`, `read_drec_spin()`, `reset_experiment_info()`, `fill_data()`, `fill_data_envelope()`, `fill_discontinuous_data()`, `fill_mode_data()`, `spin_data()`, `spin_data_pix()`, `sweep_data()`, `sweep_discontinuous_data()` and `sweep_mode_data()` |
DESCRIPTION OF THE IDFS PROGRAMMERS MANUAL

The IDFS Programmers Manual describes the set of routines that can be used to access data that is stored in IDFS format and to access the derived data products defined within an SCF file. This manual consists of ten sections entitled 1R, 2R, 3R, 4R, 1H, 2H, 3H, 4H, 1S and 3S. The section entitled 1R contains a detailed description of the basic set of IDFS data retrieval routines that return data one sample set at a time or one spin at a time. The section entitled 2R contains a detailed description of the IDFS routines that are used to retrieve data that is time-averaged, sample-averaged, or spin-averaged. Time-averaged data refers to data that is acquired for a specified time interval. Sample-averaged data refers to data that is averaged over a specific number of data samples. Spin-averaged data refers to data that is averaged over a complete spin. The section entitled 3R contains a detailed description of the basic set of SCF output retrieval routines that return data for each iteration of the SCF algorithm. The section entitled 4R contains a detailed description of the SCF routines that are used to retrieve derived data products that are time-averaged or sample-averaged. Time-averaged SCF data has the same meaning as time-averaged IDFS data. Sample-averaged SCF data refers to data that is averaged over a specific number of iterations of the SCF algorithm.

The sections entitled 1H, 2H, 3H and 4H contain a description of each of the IDFS/SCF include files. These include files contain the return codes for the IDFS/SCF routines, mnemonics that should be utilized for some of the parameter values to the various routines and the prototypes for the IDFS/SCF routines. The section entitled 1S contains a detailed description of the data structure which holds the pertinent information returned by the IDFS read routine. The last section, 3S, contains a detailed description of the data structure which holds the values returned from the execution of the algorithm in the named SCF file.

The SCF software supports post acquisition analysis. The IDFS software supports real-time and post acquisition analysis. The processing for real-time and post acquisition analysis differs somewhat based upon the nature of the data files. In the real-time scenario, the header and data files are incomplete and it is possible to attempt to read from either file prior to the data being received. This will result in a premature end-of-file (eof) and the IDFS routine will return the status code EOF_STATUS for a header file read and DREC_EOF_NO_SENSOR or DREC_EOF_SENSOR for a data file read. For real-time processing, if any of the three codes are returned, the processing simply continues, anticipating that the data will eventually be received and processed at a later date. A true end-of-file status is acknowledged by the read_drec routine, returning the status code LOS_STATUS or NEXT_FILE_STATUS, as defined in the description of the read_drec routine. For post analysis acquisition, the IDFS routines may return the code FILL_HEADER, indicating that a dummy or fill header was read from the header file - the header record was never received and placed into the file. Since the data file should be complete, no end-of-file (eof) should be returned for a call to the read routine. If an eof is encountered (zero bytes read from the file), the error code DREC_READ_ERROR is returned. If either of these return codes is encountered in post
analysis acquisition, the system should be terminated, indicating a possibly corrupted header or data file. The only end-of-file status acceptable for post analysis acquisition is the status code NEXT_FILE_STATUS or LOS_STATUS. It is also possible that a partial read may take place - that is, the number of bytes read did not match the number of bytes asked for. The IDFS routine will return the status code PARTIAL_READ in a playback scenario and EOF_STATUS in a real-time scenario and will re-position the file pointer at the start of the record in question. For real-time processing, if EOF_STATUS is returned, the processing simply continues, anticipating that the next read will result in a complete record. For post analysis acquisition, if PARTIAL_READ is returned, the system should be terminated, indicating a possibly corrupted header or data file.

All of the IDFS and SCF routines are detailed in depth within this manual. In the ARGUMENTS section of the description, a brief explanation of each argument or parameter is given. In some cases, a mnemonic is shown in parentheses for specific values. It is recommended that the user use the mnemonics instead of the specified value for the parameters. These mnemonics are defined in the user_defs.h include file for the IDFS routines and in the SCF_defs.h include file for the SCF routines. These files are described in sections 1H and 3H of the IDFS Programmers Manual. Using the mnemonics increases readability and guards against possible future changes to parameter values.

The IDFS and SCF software is written in the C programming language. In order to ease the task of porting the software to different platforms, the software utilizes typedefs. These typedefs are defined in the SDDAS_types.h include file, which can be found in section 1H of the IDFS Programmers Manual. All IDFS and SCF routines utilize the typedefs for arguments and return values.
PROGRAMMING EXAMPLES

In order to help explain how a programmer would go about developing a program that utilizes the IDFS routines, examples of programs that retrieve and display the data on the screen from the TSS-1 RETE experiment are shown below. These programs illustrate both real-time and post analysis acquisition of data for a single sensor, as well as for all sensors for a given virtual instrument.

EXAMPLE 1

```c
#include <stdio.h>
#include <string.h>
#include "ret_codes.h"
#include "user_defs.h"
#include "libCfg.h"
#include "libbase_idfs.h"
#include "libdb.h"

/* This routine processes real-time data for RTLA's sensor 0. */

void main (void)
{
    struct idf_data *EXP_DATA;
    SDDAS_FLOAT conv_data[1000];
    SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano, *tbl_oper;
    SDDAS_ULONG data_key;
    SDDAS_USHORT version;
    register SDDAS_USHORT k;
    SDDAS_SHORT rcode, sensor, ret_val;
    SDDAS_SHORT btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_CHAR extension[3], full_swp = 1, fwd = 1, *tbls_to_apply, num_tbls;
    char more_data = 1;
    void *idf_data_ptr;

    /******************************************************************************/
    /* Set the start and stop time (in this case to reflect real-time scenario), select the */
    /* sensor of interest, and select the data file of interest ("" means default file is to be */
    /* used). */
    /******************************************************************************/

    btime_yr = -1;
    btime_day = -1;
    btime_sec = -1;
    btime_nano = 0;
    etime_yr = -1;
    etime_day = -1;
```
etime_sec = -1;
etime_nano = 0;
sensor = 0;
strncpy (extension,""");
CfgInit ();
dbInitialize ();
init_idfs ();

/******************************************************************************/
/* Retrieve the key that is associated with the project, mission, experiment, instrument * /
/* and virtual instrument specified.                                        */
/******************************************************************************/
ret_val = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (ret_val != ALL_OKAY)
{
    printf ("%n Error %d from get_data_key routine.
", ret_val);
    exit (-1);
}
get_version_number (&version);

/******************************************************************************/
/* Create an instance of the idf_data structure.                           */
/******************************************************************************/
ret_val = create_idf_data_structure (&idf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf ("%n Error %d from create_idf_data_structure routine.
", ret_val);
    exit (-1);
}
EXP_DATA = (struct idf_data *) idf_data_ptr;

/******************************************************************************/
/* Open the data files associated with the time period selected for this data set /  */
/* extension / version combination.                                       */
/******************************************************************************/
ret_val = file_open (data_key, extension, version, btime_yr, btime_day, btime_sec,    
btime_nano, etime_yr, etime_day, etime_sec, etime_nano, 0);
if (ret_val != ALL_OKAY)
{
    printf ("%n Error %d from file_open routine.
", ret_val);
    exit (-1);
}
/* Since the routine FILE_OPEN sets internal flags to indicate that all sensors are to be processed, reset the flags to indicate that only sensor 0 is being requested. */

ret_val = select_sensor (data_key, extension, version, sensor);
if (ret_val != ALL_OKAY)
{
    printf("\n Error %d from select_sensor routine.\n", ret_val);
    exit (-1);
}

/* Retrieve the raw units for the data. */

num_tbls = 0;
tbls_to_apply = NULL;
tbl_oper = NULL;

/* Get the data for the requested sensor. */

while (more_data)
{
    /* Find the position in the data file closest to the requested start time for this data set. */
    /* If the file has been positioned correctly, future calls to this routine just return the ALL_OKAY status; otherwise, the routine keeps trying to read from the files and to position the file pointers (records may not have been written to disk yet). */
    ret_val = file_pos (data_key, extension, version, idf_data_ptr, btime_yr, btime_day, btime_sec, btimeNano, etime_yr, etime_day, etime_sec, etimeNano);
    if (ret_val == LOS_STATUS)
        more_data = 0;
    else if (ret_val == NEXT_FILE_STATUS)
    {
        /* For realtime processing, btime_sec is set to –1 so that when files are crossed, the routines will position the file at the beginning of that next file. */
        rcode = reset_experiment_info (data_key, extension, version, -1,-1, -1, -1, etime_yr, etime_day, etime_sec, etimeNano);
    
    /* Move to next data file if more_data must be set to zero. */
if (rcode != ALL_OKAY)
{
    printf("\nError %d from reset_experiment_info.\n", rcode);
    exit (-1);
}
}
else if (ret_val != ALL_OKAY && ret_val != EOF_STATUS)
{
    printf("\n Error %d from file_pos routine.\n", ret_val);
    exit (-1);
}

if (ret_val == ALL_OKAY)
{
    ret_val = read_drec (data_key, extension, version, idf_data_ptr, sensor, fwd, full_swp);
    if (ret_val < 0)
    {
        printf("\nError %d from read_drec.\n", ret_val);
        exit (-1);
    }
    
    /**************************************************************************/
    /* The sensor data was found within the time being processed. */
    /**************************************************************************

    if (ret_val == ALL_OKAY || EXP_DATA->filled_data)
    {
        rcode = convert_to_units (data_key, extension, version, idf_data_ptr,
                                sensor, SENSOR, 0, num_tbls, tbls_to_apply,
                                tbl_oper, conv_data, 0, 0);
        if (rcode != ALL_OKAY)
        {
            printf("\nError %d from convert_to_units.\n", rcode);
            exit (-1);
        }
        
        /**************************************************************************/
        /* Print the times for the sample being returned. */
        /**************************************************************************

        printf("\n\nSENSOR %d's START TIME_MS = %ld", sensor, EXP_DATA->bmilli);
        printf("\nSENSOR %d's START TIME_NS = %ld", sensor, EXP_DATA->bnano);
        printf("\nSENSOR %d's END TIME_MS = %ld", sensor, EXP_DATA->emilli);
        printf("\nSENSOR %d's END TIME_NS = %ld", sensor, EXP_DATA->enano);
        
    }
/****************** Print the data, 6 values per row, in exponential format. ******************/

for (k = 0; k < EXP_DATA->num_sample; ++k)
{
    if (k % 6 == 0)
        printf ("n");
    printf ("%10.2e ", conv_data[k]);
}

printf ("\n\n");

if (ret_val == LOS_STATUS)
    more_data = 0;
else if (ret_val == NEXT_FILE_STATUS)
{
    /*************************************************************/
    /* For realtime processing, btime_sec is set to -1 so that when files are crossed, the routines will position the file at the beginning of that next file. */
    /*************************************************************/

    rcode = reset_experiment_info (data_key, extension, version, -1, -1, -1, -1,
                                etime_yr, etime_day, etime_sec, etime_nano);

    if (rcode != ALL_OKAY)
    {
        printf ("Error %d from reset_experiment_info\n", rcode);
        exit (-1);
    }
}
free_experiment_info();
EXAMPLE 2

```c
#include <stdio.h>
#include <string.h>
#include "ret_codes.h"
#include "user_defn.h"
#include "libbase_idfs.h"
#include "libVIDF.h"
#include "libCfg.h"
#include "libdb.h"

/*  This routine processes real-time data for all RTLA sensors. */

void main (void)
{
    struct idf_data *EXP_DATA;
    register SDDAS_USHORT k;
    SDDAS_FLOAT conv_data[1000];
    SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano, rval, *tbl_oper;
    SDDAS_UULONG data_key;
    SDDAS_USHORT version;
    SDDAS_SHORT rcode, sensor, ret_val, num_sensor;
    SDDAS_SHORT btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_CHAR extension[3], full_swp = 1, fwd = 1, *tbls_to_apply, num_tbls;
    char more_data = 1;
    void *idf_data_ptr;

    /***************************************************************************/
    /*  Set the start and stop time (in this case to reflect real-time scenario) and select the   */
    /*  data file of interest ("" means default file is to be used). */
    /***************************************************************************/

    btime_yr = -1;
    btime_day = -1;
    btime_sec = -1;
    btime_nano = 0;

    etime_yr = -1;
    etime_day = -1;
    etime_sec = -1;
    etime_nano = 0;
    strcpy (extension,"");
    CfgInit ();
    dbInitialize ();
    init_idfs ();
```
/************************** Retrieve the key that is associated with the project, mission, experiment, instrument * /
/* and virtual instrument specified. */
/**************************/

ret_val = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from get_data_key routine.\n", ret_val);
    exit (-1);
}

get_version_number (&version);

/************************** Create an instance of the idf_data structure. */
/**************************/

ret_val = create_idf_data_structure (&idf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from create_idf_data_structure routine.\n", ret_val);
    exit (-1);
}

EXP_DATA = (struct idf_data *) idf_data_ptr;

/************************** Open the data files associated with the time period selected for this data set / extension / version combination. */
/**************************/

ret_val = file_open (data_key, extension, version, btime_yr, btime_day, btime_sec, btime_nano, etime_yr, etime_day, etime_sec, etime_nano, 0);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from file_open routine.\n", ret_val);
    exit (-1);
}

/************************** Find out the number of sensors defined for this virtual instrument. */
/**************************/

rval = read_idf (data_key, extension, version, (SDDAS_CHAR *) &num_sensor, _SEN, 0, 0, 1);
if (rval < 0)
{
    printf ("\n Error %ld from read_idf routine.\n", rval);
    exit (-1);
}

/**************************************************************/
/* Retrieve the raw units for the data.                      */
/**************************************************************/

num_tbls = 0;
tbls_to_apply = NULL;
tbl_oper = NULL;

/**************************************************************/
/* Get the data for all sensors.                            */
/**************************************************************/

while (more_data)
{
    /******************************************************************************/
    /* Find the position in the data file closest to the requested start time for this data set. */
    /* If the file has been positioned correctly, future calls to this routine just return the   */
    /* ALL_OKAY status; otherwise, the routine keeps trying to read from the files and            */
    /* to position the file pointers (records may not have been written to disk yet).            */
    /******************************************************************************/

    ret_val = file_pos (data_key, extension, version, idf_data_ptr, btime_yr, btime_day,
                        btime_sec, btime_nano, etime_yr, etime_day, etime_sec, etime_nano);
    if (ret_val == LOS_STATUS)
        more_data = 0;
    else if (ret_val == NEXT_FILE_STATUS)
    {
        /******************************************************************************/
        /* For realtime processing, btime_sec is set to -1 so that when files are            */
        /* crossed, the routines will position the file at the beginning of that next file.    */
        /******************************************************************************/

        rcode = reset_experiment_info (data_key, extension, version, -1, -1, -1, -1,
                                       etime_yr, etime_day, etime_sec, etime_nano);
        if (rcode != ALL_OKAY)
        {
            printf ("\nError %d from reset_experiment_info.\n", rcode);
            exit (-1);
        }
    }
else if (ret_val != ALL_OKAY && ret_val != EOF_STATUS)
{
    printf ("\n Error %d from file_pos routine.\n", ret_val);
    exit (-1);
}
if (ret_val == ALL_OKAY)
{
    for (sensor = 0; sensor < num_sensor; ++sensor)
    {
        /*********************************************************************/
        /* Advance to the next data set only if the last sensor is being processed to ensure */
        /* all samples which occur at the same time are processed simultaneously. */
        /*********************************************************************/

        fwd = (sensor == num_sensor - 1) ? 1 : 0;
        ret_val = read_drec (data_key, extension, version, idf_data_ptr, sensor, fwd, full_swp);
        if (ret_val < 0)
        {
            printf ("\nError %d from read_drec.\n", ret_val);
            exit (-1);
        }
        /*********************************************************************/
        /* The sensor data was found within the time being processed. */
        /*********************************************************************/

        if (ret_val == ALL_OKAY || EXP_DATA->filled_data)
        {
            rcode = convert_to_units (data_key, extension, version, idf_data_ptr, sensor,
                                      SENSOR, 0, num_tbls, tbls_to_apply, tbl_oper,
                                      conv_data, 0, 0);
            if (rcode != ALL_OKAY)
            {
                printf ("\nError %d from convert_to_units.\n", rcode);
                exit (-1);
            }
            /*********************************************************************/
            /* Print the times for the sample being returned. */
            /*********************************************************************/

            printf ("\n\nSENSOR %d's START TIME_MS = %ld", sensor, EXP_DATA->bmilli);
            printf ("\nSENSOR %d's START TIME_NS = %ld", sensor, EXP_DATA->bnano);
            printf ("\nSENSOR %d's END TIME_MS = %ld", sensor, EXP_DATA->emilli);
printf ("\nSENSOR %d's END TIME_NS = %ld", sensor, EXP_DATA->enano);

/***************************************************************/
/* Print data values, 6 values per row, in exponential format. */
/***************************************************************/

for (k = 0; k < EXP_DATA->num_sample; ++k)
{
    if (k % 6 == 0)
        printf ("\n");
    printf ("%10.2e  ", conv_data[k]);
}

printf ("\n\n");

if (ret_val == LOS_STATUS)
    more_data = 0;
else if (ret_val == NEXT_FILE_STATUS)
{
    /******************************************************************************
    /* For realtime processing, btime_sec is set to -1 so that when files are     */
    /* crossed, the routines will position the file at the beginning of that next file. */
    /******************************************************************************

    rcode = reset_experiment_info (data_key, extension, version, -1, -1, -1,
                                    etime_yr, etime_day, etime_sec, etime_nano);
    if (rcode != ALL_OKAY)
    {
        printf ("\nError %d from reset_experiment_info.\n", rcode);
        exit (-1);
    }
}
free_experiment_info();
}
EXAMPLE 3

#include <stdio.h>
#include <string.h>
#include "ret_codes.h"
#include "user_defs.h"
#include "libbase_idfs.h"
#include "libCfg.h"
#include "libdb.h"

/*  This routine processes playback data for RTLA's sensor 0. */

void main (void)
{
    struct idf_data *EXP_DATA;
    SDDAS_FLOAT conv_data[1000];
    SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano, ret_time_sec;
    SDDAS_LONG ret_time_nano, new_start_sec, new_start_nsec, *tbl_oper;
    SDDAS_ULONG data_key;
    SDDAS_USHORT version;
    register SDDAS_USHORT k;
    SDDAS_SHORT sensor, ret_val, rcode, new_year, new_day;
    SDDAS_SHORT btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_CHAR extension[3], full_swp = 1, fwd = 1, *tbls_to_apply, num_tbls;
    char more_data = 1;
    void *idf_data_ptr;

    /**************************************************************************/
    /*  Set the start and stop time. This example uses year 1991, day 93, starting at 00:01:21  */
    /*  (hh:mm:ss) - 81 seconds - and ending at 00:25:36 (hh:mm:ss) - 1536 seconds.          */
    /**************************************************************************/

    btime_yr = 1991;
    btime_day = 93;
    btime_sec = 81;
    btime_nano = 0;

    etime_yr = 1991;
    etime_day = 93;
    etime_sec = 1536;
    etime_nano = 0;

    /**************************************************************************/
    /*  Set the sensor of interest and the data file of interest ("" means default file is to be used).*/
    /**************************************************************************/
sensor = 0;
strcpy (extension,"");
CfgInit ();
dbInitialize ();
init_idfs ();

/*********************************************************************/
/*  Retrieve the key that is associated with the project, mission, experiment, instrument  */
/*  and virtual instrument specified.                                                                                    */
/*********************************************************************/
ret_val = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (ret_val != ALL_OKAY)
{
  printf ("n Error %d from get_data_key routine.\n", ret_val);
  exit (-1);
}
get_version_number (&version);

/*********************************************************************/
/*  Create an instance of the idf_data structure.                   */
/*********************************************************************/
ret_val = create_idf_data_structure (&idf_data_ptr);
if (ret_val != ALL_OKAY)
{
  printf ("n Error %d from create_idf_data_structure routine.\n", ret_val);
  exit (-1);
}
EXP_DATA = (struct idf_data *) idf_data_ptr;

/*********************************************************************/
/*  Open the data files associated with the time period selected for this data set / */
/*  extension/version combination.                                                                                  */
/*********************************************************************/
ret_val = file_open (data_key, extension, version, btime_yr, btime_day, btime_sec,
  btime_nano, etime_yr, etime_day, etime_sec, etime_nano, 0);
if (ret_val != ALL_OKAY)
{
  printf ("n Error %d from file_open routine.\n", ret_val);
  exit (-1);
}
/*********************************************************************/
/*  Since the routine FILE_OPEN sets internal flags to indicate that all sensors are to */
/*  be processed, reset the flags to indicate that only sensor 0 is being requested.    */
/*********************************************************************/

ret_val = select_sensor (data_key, extension, version, sensor);
if (ret_val != ALL_OKAY)
{
    printf ("n Error %d from select_sensor routine.\n", ret_val);
    exit (-1);
}

/*********************************************************************/
/*  Find the position in the data file closest to the requested start time for this data set.  */
/*  Unlike real-time, if an error is encountered, the system should terminate - no need to */
/*  retry in anticipation of incoming data.                                          */
/*********************************************************************/
ret_val = file_pos (data_key, extension, version, idf_data_ptr, btime_yr, btime_day,
    btime_sec, btime_nano, etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf ("n Error %d from file_pos.\n", ret_val);
    exit (-1);
}

/*********************************************************************/
/*  Retrieve the raw units for the data.                                         */
/*********************************************************************/

num_tbls = 0;
tbls_to_apply = NULL;
tbl_oper = NULL;

/*********************************************************************/
/*  Get the data for the requested sensor.  Terminate when the requested end time is reached*/
/*  or when no more data files are available for processing.                      */
/*********************************************************************/

while (more_data)
{
    ret_val = read_drec (data_key, extension, version, idf_data_ptr, sensor, fwd, full_swp);
    if (ret_val < 0)
    {
        printf ("nError %d from read_drec.\n", ret_val);
    }
exit (-1);
}

/* The sensor data was found within the time sample being processed. */

if (ret_val == ALL_OKAY || EXP_DATA->filled_data)
{
    ret_time_sec = (EXP_DATA->bmilli + (EXP_DATA->bnano / 1000000)) / 1000;
    ret_time_nano = (EXP_DATA->bmilli % 1000) * 1000000 + EXP_DATA->bnano;

    /* The requested end time has been reached? */
    if (EXP_DATA->byear > etime_yr ||
        (EXP_DATA->byear == etime_yr && EXP_DATA->bday > etime_day) ||
        (EXP_DATA->byear == etime_yr && EXP_DATA->bday == etime_day &&
         ret_time_sec > etime_sec) ||
        (EXP_DATA->byear == etime_yr && EXP_DATA->bday == etime_day &&
         ret_time_sec == etime_sec && ret_time_nano > etime_nano))
    {
        more_data = 0;
        break;
    }
}

rcode = convert_to_units (data_key, extension, version, idf_data_ptr, sensor,
        SENSOR, 0, num_tbls, tbls_to_apply, tbl_oper, conv_data,
        0, 0);
if (rcode != ALL_OKAY)
{
    printf ("\nError %d from convert_to_units.\n", rcode);
    exit (-1);
}

/* Print the times for the sample being returned. */

printf("\n\nSENSOR %d's START TIME_MS = %ld", sensor, EXP_DATA->bmilli);
printf("\nSENSOR %d's START TIME_NS = %ld", sensor, EXP_DATA->bnano);
printf("\nSENSOR %d's END TIME_MS = %ld", sensor, EXP_DATA->emilli);
printf("\nSENSOR %d's END TIME_NS = %ld", sensor, EXP_DATA->enano);
/*********************************************************************/
/*  Print the data values, 6 values per row, in exponential format.  */
/*********************************************************************/
for (k = 0; k < EXP_DATA->num_sample; ++k)
{
    if (k % 6 == 0)
        printf("\n");
    printf("%10.2e  ", conv_data[k]);
}

printf("\n\n");

if (ret_val == LOS_STATUS || ret_val == NEXT_FILE_STATUS)
{
    /***********************************************************************/
    /*   Get the start time to use to get the next data file.              */
    /***********************************************************************/

    rcode = next_file_start_time (data_key, extension, version, 0, &new_year,
                                &new_day, &new_start_sec, &new_start_nsec);
    if (rcode != ALL_OKAY)
    {
        printf("\n Error %d from next_file_start_time.\n", rcode);
        exit (-1);
    }

    rcode = reset_experiment_info (data_key, extension, version, new_year,
                                   new_day, new_start_sec, new_start_nsec,
                                   etime_yr, etime_day, etime_sec, etime_nano);
    if (rcode == NO_DATA)
    {
        more_data = 0;
        break;
    }
    else if (rcode != ALL_OKAY)
    {
        printf("\n Error %d from reset_experiment_info.\n", rcode);
        exit (-1);
    }

    rcode = file_pos (data_key, extension, version, idf_data_ptr, new_year, new_day,
                      new_start_sec, new_start_nsec, etime_yr, etime_day, etime_sec,
                      etime_nano);

}
if (rcode != ALL_OKAY)
{
    printf ("\n Error %d from file_pos.\n", rcode);
    exit (-1);
}

free_experiment_info();
EXAMPLE 4

#include <stdio.h>
#include <string.h>
#include "ret_codes.h"
#include "user_defns.h"
#include "libbase_idfs.h"
#include "libVIDF.h"
#include "libCfg.h"
#include "libdb.h"

/* This routine processes playback data for all RTLA sensors. */

void main (void)
{
    struct idf_data *EXP_DATA;
    SDDAS_FLOAT conv_data[1000];
    SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano, ret_time_sec;
    SDDAS_LONG ret_time_nano, new_start_sec, new_start_nsec, rval, *tbl_oper;
    SDDAS_ULONG data_key;
    SDDAS_USHORT version;
    register SDDAS_USHORT k;
    SDDAS_SHORT sensor, ret_val, num_sensor, rcode, new_year, new_day;
    SDDAS_SHORT btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_CHAR extension[3], full_swp = 1, fwd = 1, *tbls_to_apply, num_tbls;
    char more_data = 1;
    void *idf_data_ptr;

    /**************************************************************************/
    /* Set the start and stop time. This example uses year 1991, day 93, starting at 00:01:21 (hh:mm:ss) - 81 seconds - ending at 00:25:36 (hh:mm:ss) - 1536 seconds. */
    /**************************************************************************/
    btime_yr = 1991;
    btime_day = 93;
    btime_sec = 81;
    btime_nano = 0;
    etime_yr = 1991;
    etime_day = 93;
    etime_sec = 1536;
    etime_nano = 0;

    /**************************************************************************/
    /* Set the data file of interest ("" means default file is to be used). */
    /**************************************************************************/
strcpy (extension,"");  
CfgInit ();  
dblInitialize ();  
init_idfs ();  

/*******************************************************************************/  
/* Retrieve the key that is associated with the project, mission, experiment, instrument */  
/* and virtual instrument specified. */  
/*******************************************************************************/  
ret_val = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);  
if (ret_val != ALL_OKAY)  
  
  printf ("\n Error %d from get_data_key routine.\n", ret_val);  
  exit (-1);  
}  
get_version_number (&version);  

/*******************************************************************************/  
/* Create an instance of the idf_data structure. */  
/*******************************************************************************/  
ret_val = create_idf_data_structure (&idf_data_ptr);  
if (ret_val != ALL_OKAY)  
  
  printf ("\n Error %d from create_idf_data_structure routine.\n", ret_val);  
  exit (-1);  
}  
EXP_DATA = (struct idf_data *) idf_data_ptr;  

/*******************************************************************************/  
/* Open the data files associated with the time period selected for this data set */  
/* extension / version combination. */  
/*******************************************************************************/  
ret_val = file_open (data_key, extension, version, btime_yr, btime_day, btime_sec,  
  btime_nano, etime_yr, etime_day, etime_sec, etime_nano, 0);  
if (ret_val != ALL_OKAY)  
  
  printf ("\n Error %d from file_open routine.\n", ret_val);  
  exit (-1);  
}  

/*******************************************************************************/  
/* Find the position in the data file closest to the requested start time for this data set */  
/*******************************************************************************/
ret_val = file_pos (data_key, extension, version, idf_data_ptr, btime_yr, btime_day,
    btime_sec, btime_nano, etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
    {
        printf ("\n Error %d from file_pos.\n", ret_val);
        exit (-1);
    }

rval = read_idf (data_key, extension, version, (SDDAS_CHAR *) &num_sensor,
    _SEN, 0, 0, 1);
if (rval < 0)
    {
        printf ("\n Error %ld from read_idf routine.\n", rval);
        exit (-1);
    }

while (more_data)
    {
        for (sensor = 0; sensor < num_sensor; ++sensor)
            {
                num_tbls = 0;
                tbls_to_apply = NULL;
                tbl Oper = NULL;

                /* Advance to the next data set only if the last sensor is being processed to ensure */
                /* all samples which occur at the same time are processed simultaneously. */
            }
fwd = (sensor == num_sensor - 1) ? 1 : 0;
ret_val = read_drec (data_key, extension, version, idf_data_ptr, sensor, fwd,
full_swp);
if (ret_val < 0)
{
    printf ("\nError %d from read_drec.\n", ret_val);
    exit (-1);
}

/***************************************************************************/
/* The sensor data was found within the time being processed. */
/***************************************************************************/

if (ret_val == ALL_OKAY || EXP_DATA->filled_data)
{
    /***************************************************************************/
    /* If the time of the sample is past the requested end time, stop processing data. */
    /***************************************************************************/
    ret_time_sec = (EXP_DATA->bmilli + (EXP_DATA->bnano / 1000000)) / 1000;
    ret_time_nano = (EXP_DATA->bmilli % 1000) * 1000000 + EXP_DATA->bnano;
    if (EXP_DATA->byear > etime_yr ||
        (EXP_DATA->byear == etime_yr &&
         EXP_DATA->bday > etime_day) ||
        (EXP_DATA->byear == etime_yr &&
         EXP_DATA->bday == etime_day &&
         ret_time_sec > etime_sec) ||
        (EXP_DATA->byear == etime_yr &&
         EXP_DATA->bday == etime_day &&
         ret_time_sec == etime_sec &&
         ret_time_nano > etime_nano))
    {  
        more_data = 0;
        break;
    }
    rcode = convert_to_units (data_key, extension, version, idf_data_ptr, sensor,
        SENSOR, 0, num_tbls, tbls_to_apply, tbl_oper,
        conv_data, 0, 0);
    if (rcode != ALL_OKAY)
    {
        printf ("\nError %d from convert_to_units.\n", rcode);
        exit (-1);
    }
/***************************************************************************/
/* Print the times for the sample being returned. */
/******************************************************************************/

24
printf ("\n\nSENSOR %d's START TIME_MS = %ld", sensor, EXP_DATA->bmilli);
printf ("\nSENSOR %d's START TIME_NS = %ld", sensor, EXP_DATA->bnano);
printf ("\nSENSOR %d's END TIME_MS = %ld", sensor, EXP_DATA->emilli);
printf ("\nSENSOR %d's END TIME_NS = %ld", sensor, EXP_DATA->enano);

/**************************************************************/
/*  Print the data, 6 values per row, in exponential format.                          */
/**************************************************************/
for (k = 0; k < EXP_DATA->num_sample; ++k)
{
    if (k % 6 == 0)
        printf ("\n");
    printf ("%10.2e  ", conv_data[k]);
}
printf ("\n\n");

if (ret_val == LOS_STATUS || ret_val == NEXT_FILE_STATUS)
{
    /**********************************************************************/
    /*  Get the start time to use to get the next data file.               */
    /**********************************************************************/
    rcode = next_file_start_time (data_key, extension, version, 0, &new_year,
                                &new_day, &new_start_sec, &new_start_nsec);
    if (rcode != ALL_OKAY)
    {
        printf ("\n Error %d from next_file_start_time.\n", rcode);
        exit (-1);
    }
    rcode = reset_experiment_info (data_key, extension, version, new_year,
                                  new_day, new_start_sec, new_start_nsec,
                                  etime_yr, etime_day, etime_sec, etimeNano);
    if (rcode == NO_DATA)
    {
        more_data = 0;
        break;
    }
    else if (rcode != ALL_OKAY)
    {
        printf ("\n Error %d from reset_experiment_info.\n", rcode);
        exit (-1);
    }
rcode = file_pos (data_key, extension, version, idf_data_ptr, new_year, new_day,
    new_start_sec, new_start_nsec, etime_yr, etime_day, etime_sec, etime_nano);
    if (rcode != ALL_OKAY)
    {
        printf ("\n Error %d from file_pos.\n", rcode);
        exit (-1);
    }
}
free_experiment_info();
EXAMPLE 5

This example demonstrates the usage of most of the IDFS routines. The example was coded for real-time, processing all sensors for the virtual instrument in question. The user can refer to the previous coding examples to determine how to change this example to process post-time data or single sensor data only.

```c
#include <stdio.h>
#include <string.h>
#include "ret_codes.h"
#include "user_defs.h"
#include "libtrec_idfs.h"
#include "libVIDF.h"
#include "libCfg.h"
#include "libdb.h"

/*  This routine processes real-time data for all RTLA sensors.  */

void main (void)
{
    struct idf_data *EXP_DATA;
    register SDDAS_FLOAT *dptr, *frac;
    register SDDAS_SHORT loop, i;
    SDDAS_UULONG data_key;
    SDDAS_USHORT vnum;
    SDDAS_FLOAT *ret_data, *ret_frac, sen_min, sen_max, *base_data, *base_frac;
    SDDAS_FLOAT *center_ptr, *band_low, *band_high, actual_phi, *data_ptr;
    SDDAS_FLOAT start_range[6], stop_range[6];
    SDDAS_LONG btime_sec, btime_nsec, etime_sec, etime_nsec, base_sec, base_nano, base_pix;
    SDDAS_LONG res_sec, res_nano, *start_time_sec, *start_time_nano, offset_buf, rval;
    SDDAS_LONG *end_time_sec, *end_time_nano, *bpix, *epix, offset_unit, tbl_oper[2];
    SDDAS_SHORT sensor, ret_val, accum_bin_stat, num_sensor, *sen_numbers;
    SDDAS_SHORT *num_units, data_block, uind_raw, uind_base, buf_num, sen_units;
    SDDAS_SHORT num_bands, num_converted, rcode, fill_code, num_sen;
    SDDAS_SHORT btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_SHORT *start_yr, *start_day, *end_yr, *end_day;
    SDDAS_CHAR extension[3], data_type, hdr_change, num_tbls, tbls_to_apply[2];
    SDDAS_CHAR *buf_stat, *ret_bin, *bin_stat, *base_bin, last_plot, num_center_band;
    SDDAS_CHAR dimen_status[6];
    char more_data = 1, first_time = 1;
    void *idf_data_ptr;
```
/************************** Set the start and stop time (in this case to reflect real-time scenario) and select the data file of interest ('"" means default file is to be used). *****************************/

btime_yr = -1;
btime_day = -1;
btime_sec = -1;
btime_nsec = -1;
etime_yr = -1;
etime_day = -1;
etime_sec = -1;
etime_nsec = -1;
strcpy (extension,"");CfgInit();
dbInitialize();
init_idfs();

/************************** Retrieve the key that is associated with the project, mission, experiment, instrument and virtual instrument specified. *****************************/

ret_val = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (ret_val != ALL_OKAY)
{
    printf ("n Error %d from get_data_key routine.\n", ret_val);
    exit (-1);
}
get_version_number (&vnum);

ret_val = create_idf_data_structure (&idf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf ("n Error %d from create_idf_data_structure routine.\n", ret_val);
    exit (-1);
}
EXP_DATA = (struct idf_data *) idf_data_ptr;

/************************** The data will be collapsed over the scan dimension to squash the data over the entire frequency range for this virtual instrument. *****************************/
dimen_status[0] = DIMEN_ON;
start_range[0] = 0.16;
stop_range[0] = 0.9;
start_range[1] = stop_range[1] = 0.0;
start_range[2] = stop_range[2] = 0.0;
start_range[3] = stop_range[3] = 0.0;
start_range[4] = stop_range[4] = 0.0;
start_range[5] = stop_range[5] = 0.0;
dimen_status[1] = DIMEN_OFF;
dimen_status[2] = DIMEN_OFF;
dimen_status[3] = DIMEN_OFF;
dimen_status[4] = DIMEN_OFF;
dimen_status[5] = DIMEN_OFF;

ret_val = file_open (data_key, extension, vnum, btime_yr, btime_day, btime_sec,
        btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec, 0);
if (ret_val != ALL_OKAY)
       {
            printf ("n Error %d from file_open routine.\n", ret_val);
            exit (-1);
       }

rval = read_idf (data_key, extension, vnum, (SDDAS_CHAR *) &num_sensor, _SEN, 0,
        0, 1);
if (rval < 0)
       {
            printf ("n Error %ld from read_idf routine.\n", rval);
            exit (-1);
       }

while (more_data)
       {

}
/*************** Find the position in the data file closest to the requested start time for this data set.*************/
/* If the file has been positioned correctly, future calls to this routine just return the */
/* ALL_OKAY status; otherwise, the routine keeps trying to read from the files */
/* and to position the file pointers (records may not have been written to disk yet). */
/*********************************************************************/
rcode = file_pos (data_key, extension, vnum, idf_data_ptr, btime_yr, btime_day,
             btime_sec, btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec);
if (rcode == LOS_STATUS)
  more_data = 0;
else if (rcode == NEXT_FILE_STATUS)
{
  /*************/
  /* For realtime processing, btime_sec is set to -1 so that when files are crossed, */
  /* the routines will position the file at the beginning of that next file. */
  /*************/
  ret_val = reset_experiment_info (data_key, extension, vnum, -1,-1, -1, -1,
                  etime_yr, etime_day, etime_sec, etime_nsec);
  if (ret_val != ALL_OKAY)
  {
    printf ("\nError %d from reset_experiment_info.\n", ret_val);
    exit (-1);
  }
  } else if (rcode != ALL_OKAY && rcode != EOF_STATUS)
  {
    printf("\n Error %d from file_pos routine.\n", rcode);
    exit (-1);
  }
  /*************/
  /* Some items need to be set just once. */
  /*************/
else if (first_time)
{
  first_time = 0;
  /*************/
  /* Set the base reference time, location and duration for the data buffers. */
  /*************/
  ret_val = read_drec (data_key, extension, vnum, idf_data_ptr, 0, 0, 0);
if (ret_val < 0)
{
    printf ("\nError %d from read_drec.\n", ret_val);
    exit (-1);
}

base_sec = EXP_DATA->bmilli / 1000;
base_nano = (EXP_DATA->bmilli % 1000) * 1000000 + EXP_DATA->bnano;
base_pix = 0;
res_sec = 9;
res_nano = 216000000;
set_time_values (vnum, EXP_DATA->byear, EXP_DATA->bday, base_sec,
    base_nano, base_pix, res_sec, res_nano);

/*****************************************************************************/
/*  Select sensor data in raw units for all sensors.                          */
/*****************************************************************************/

num_tbls = 0;
data_type = SENSOR;
sen_min = VALID_MIN;
sen_max = VALID_MAX;

for (sensor = 0; sensor < num_sensor; ++sensor)
{
    ret_val = fill_sensor_info (data_key, extension, vnum, sensor, sen_min, sen_max, num_tbls,
        tbls_to_apply, tbl_oper, data_type, 0);
    if (ret_val != ALL_OKAY)
    {
        printf ("\nError %d from fill_sensor_info.\n", ret_val);
        exit (-1);
    }
}

/*****************************************************************************/
/*  Select sensor data in base units for all sensors.                        */
/*****************************************************************************/

num_tbls = 2;
tbls_to_apply[0] = 0;
tbls_to_apply[1] = 1;
tbl_oper[0] = 0;
tbl_oper[1] = 3;
for (sensor = 0; sensor < num_sensor; ++sensor)
{
    ret_val = fill_sensor_info (data_key, extension, vnum, sensor, sen_min, sen_max, num_tbls,
        tbls_to_apply, tbl_oper, data_type, 0);
    if (ret_val != ALL_OKAY)
    {
        printf("\nError %d from fill_sensor_info.\n", ret_val);
        exit (-1);
    }
}

致

/*****************************/
/* Specify that 8 linear center bins from 0.16 to 0.86 (.1 delta) are to be created       */
/* and missing bins are to be left empty. No need to call SET_SCAN_INFO since */
/* raw sweep step values are desired for this sweeping instrument. Since            */
/* num_center_band is 0, the contents of tbl_oper and tbls_to_apply don't matter. */
/*****************************/
num_center_band = 0;
ret_val = set_bin_info (data_key, extension, vnum, VARIABLE_SWEEP,
    0.16, 0.86, 0.1, 8, LIN_SPACING, num_center_band,
    tbls_to_apply, tbl_oper, num_center_band,
    tbls_to_apply, tbl_oper, num_center_band,
    tbls_to_apply, tbl_oper, 'L', POINT_STORAGE, NO_BIN_FILL);
if (ret_val != ALL_OKAY)
{
    printf("\nError %d from set_bin_info.\n", ret_val);
    exit (-1);
}

/*****************************/
/* The scan units should be in terms of frequency.                                */
/*****************************/
num_tbls = 1;
tbls_to_apply[0] = 0;
tbl_oper[0] = 0;
ret_val = set_scan_info (data_key, extension, vnum, num_tbls, tbls_to_apply, tbl_oper);
if (ret_val != ALL_OKAY)
{
    printf("\nError %d from set_scan_info.\n", ret_val);
    exit (-1);
}
sensor = 0;
ret_val = center_and_band_values (data_key, extension, vnum, idf_data_ptr, sensor, 
1, 1, &center_ptr, &band_low, &band_high, 
&num_bands, &num_converted);

if (ret_val != ALL_OKAY && ret_val != CENTER_CONVERSION)
{
    printf ("%d from center_and_band_values.
", ret_val);
    exit (-1);
}

ret_val = set_collapse_info (data_key, extension, vnum, 2, 360.0, &actual_phi, 0);
if (ret_val != ALL_OKAY)
{
    printf ("%d from set_collapse_info.
", ret_val);
    exit (-1);
}

if (rcode == ALL_OKAY)
{
    fill_code = fill_data (data_key, extension, vnum, idf_data_ptr, &sen_numbers, &ret_data, 
&ret_frac, &ret_bin, &bpix, &epix, &buf_stat, &numSen, 
&num_units, &data_block, &start_yr, &start_day, &start_time_sec, 
&start_time_nano, &end_yr, &end_day, &end_time_sec, 
&end_time_nano, &hdr_change, 255);
    if (fill_code != ALL_OKAY && fill_code != LOS_STATUS && fill_code != EOF_STATUS && fill_code != NEXT_FILE_STATUS)
    {
        printf ("%d from fill_data.
", fill_code);
        exit (-1);
    }
}
for (sensor = 0; sensor < num_sensor; ++sensor)
{
    for (i = 0; i < num_sen; ++i)
    {
        if (*(sen_numbers + i) == sensor)
        {
            offset_unit = *(num_units + i) * NUM_BUFFERS * data_block;
            base_data = ret_data + offset_unit;
            base_frac = ret_frac + offset_unit;
            base_bin = ret_bin + offset_unit;

            num_tbls = 0;
            ret_val = units_index (data_key, extension, vnum, sensor, sen_min, sen_max,
                                tbsls_to_apply, tbl_oper, data_type, 0, &uind_raw,
                                &sen_units, num_tbls);
            if (ret_val != ALL_OKAY)
            {
                printf ("\nError %d from units_index.\n", ret_val);
                exit (-1);
            }

            num_tbls = 2;
            tbsls_to_apply[0] = 0;
            tbsls_to_apply[1] = 1;
            tbl_oper[0] = 0;
            tbl_oper[1] = 3;

            ret_val = units_index (data_key, extension, vnum, sensor, sen_min, sen_max,
                                tbsls_to_apply, tbl_oper, data_type, 0, &uind_base,
                                &sen_units, num_tbls);
            if (ret_val != ALL_OKAY)
            {
                printf ("\nError %d from units_index.\n", ret_val);
                exit (-1);
            }
        }
    }
}
for (buf_num = 0; buf_num < NUM_BUFFERS; ++buf_num)
    if (*(buf_stat + buf_num) == BUFFER_READY)
        {
            /****************************************************/
            /*  Print the raw units data.                       */
            /****************************************************/
            offset_buf = buf_num * sen_units * data_block;
            offset_unit = uind_raw * data_block;
            dptr = base_data + offset_buf + offset_unit;
            frac = base_frac + offset_buf + offset_unit;
            bin_stat = base_bin + offset_buf + offset_unit;
            accum_bin_stat = 0;
            for (loop = 0; loop < data_block; ++loop)
                accum_bin_stat += *(bin_stat + loop);
            /****************************************************/
            /*  Make sure there is some data in the buffers.     */
            /*  Still check bin_stat since frac will be 0.0 if   */
            /*  bin_stat = 0. For this data set, it is known     */
            /*  that the sweep values are contiguous.            */
            /****************************************************/
            if (accum_bin_stat != 0)
                {
                    printf ("sensor %d information\n", sensor);
                    printf ("start_pix = %ld", *(bpix + buf_num));
                    printf ("end_pix = %ld", *(epix + buf_num));
                    for (loop = 0; loop < data_block; ++loop)
                        {
                            if (*(bin_stat + loop) != 0)
                                printf ("raw data[%d] = %10.2e from freq bin
%.2f to %.2f",loop,*(dptr+loop) / *(frac+loop),
                            *(band_low + loop) + 1));
                            else
                                printf ("raw data[%d] = %10.2e from freq bin
%.2f to %.2f",loop,*(dptr + loop),
                            *(band_low + loop) + 1));
                        }
                }
            /****************************************************/
            /*  Print the base units data.                       */
            /****************************************************/
        }
offset_unit = uind_base * data_block;
dptr = base_data + offset_buf + offset_unit;
frac = base_frac + offset_buf + offset_unit;
bin_stat = base_bin + offset_buf + offset_unit;
accum_bin_stat = 0;

for (loop = 0; loop < data_block; ++loop)
    accum_bin_stat += *(bin_stat + loop);

if (accum_bin_stat != 0)
{
    printf ("\n\n sensor %d information\n", sensor);
    printf ("\n start_pix = %ld", *(bpix + buf_num));
    printf ("\n end_pix = %ld", *(epix + buf_num));

    for (loop = 0; loop < data_block; ++loop)
    {
        if (*(bin_stat + loop) != 0)
            printf ("\n base data[%d] = %10.2e from freq bin %.2f to %.2f",*(dptr+loop)/*(frac+loop),
                    *(band_low + loop), *(band_low + loop + 1));
        else
            printf ("\n base data[%d] = %10.2e from freq bin %.2f to %.2f", *(dptr + loop),
                    *(band_low + loop), *(band_low + loop + 1));
    }
}

/***********************************************************************/
/*  Collapse the data over the scan dimension.                        */
/***********************************************************************/

for (buf_num = 0; buf_num < NUM_BUFFERS; ++buf_num)
if (*(buf_stat + buf_num) == BUFFER_READY)
{
    /***************************************************************************/
    /*  Place the data for the current buffer into the matrix used to collapse data */
    /*  over the theta and/or scan dimensions (no phi, mass or charge dimensions).*/
    /***************************************************************************/
ret_val = fill_theta_matrix (data_key, extension, vnum, num_sen,
    sen_numbers, ret_data, ret_frac,
    ret_bin, num_units, buf_num, sen_units);

if (ret_val != ALL_OKAY)
{"printf ("\nError %d from fill_theta_matrix.\n", ret_val);
exit (-1);
"
}

for (sensor = 0; sensor < num_sensor; ++sensor)
{"******************
/* Collapse over scan dimension for the raw data. The single value will */
/* be set to -3.4e38 (OUTSIDE_MIN) if no data was present in the data */
/* buffers for the sensor in question.                             */
******************

ret_val = collapse_dimensions (data_key, extension, vnum, sensor, dimen_status,
    start_range, stop_range, STRAIGHT_AVG, 0,
    &data_ptr, 0, 1, 2, 0.0, 1, 0, 0, 0, 1, buf_num);

if (ret_val != ALL_OKAY)
{"printf ("\nError %d from collapse_dimensions.\n", ret_val);
exit (-1);
"
}

if (*data_ptr > OUTSIDE_MIN)
printf ("\n data from collapsing raw units for sensor %d is %10.2e",
    sensor, *data_ptr);

******************
/* Collapse over scan dimension for the base data. The single value will */
/* be set to -3.4e38 (OUTSIDE_MIN) if no data was present in the data */
/* buffers for the sensor in question.                             */
******************

last_plot = (sensor == num_sensor - 1) ? 1 : 0;
ret_val = collapse_dimensions (data_key, extension, vnum, sensor, dimen_status,
    start_range, stop_range, STRAIGHT_AVG, 0,
    &data_ptr, 0, 1, 2, 0.0, 1, 0, 1, last_plot, 1, buf_num);

if (ret_val != ALL_OKAY)
{"printf ("\nError %d from collapse_dimensions.\n", ret_val);
exit (-1);
"}
if (*data_ptr > OUTSIDE_MIN)
    printf("n data from collapsing base units for sensor %d is %10.2e", sensor, *data_ptr);
}
}

if (fill_code == LOS_STATUS)
    more_data = 0;
else if (fill_code == NEXT_FILE_STATUS)
{
    /******************************************************************
    /* For realtime processing, btime_sec is set to -1 so that when files are crossed, */
    /* the routines will position the file at the beginning of that next file. The */
    /* mandatory call to file_pos can be found at the top of while loop. */
    /******************************************************************/

    ret_val = reset_experiment_info (data_key, extension, vnum, -1, -1, -1, -1,
                                         etime_yr, etime_day, etime_sec, etime_nsec);
    if (ret_val != ALL_OKAY)
    {
        printf("nError %d from reset_experiment_info.\n", ret_val);
        exit (-1);
    }
}

free_experiment_info();

All of the routines used within the above program are detailed in depth within sections 1R and 2R of this manual. In all IDFS coding examples provided, the database assignment names for the project, mission, experiment, instrument and virtual instrument are used to determine the data key that is to be used for the data set in question.
EXAMPLE 6

This example demonstrates how a spin’s worth of data can be acquired and processed.

```c
#include <stdio.h>
#include <string.h>
#include "ret_codes.h"
#include "user_defs.h"
#include "libtrec_idfs.h"
#include "libCfg.h"
#include "libdb.h"

void main (int argc, char **argv)
{
    struct idf_data *EXP_DATA;
    register SDDAS_FLOAT *dptr, *frac;
    register SDDAS_SHORT loop, i;
    SDDAS_ULONG data_key;
    SDDAS_USHORT vnum;
    SDDAS_FLOAT *ret_data, *ret_frac, sen_min, sen_max, *base_data, *base_frac;
    SDDAS_FLOAT *center_ptr, *band_low, *band_high;
    SDDAS_DOUBLE frac_sec;
    SDDAS_LONG btime_sec, etime_sec;
    SDDAS_LONG *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
    SDDAS_SHORT offset_unit, etime_nsec, btime_nsec, tbl_oper[5];
    SDDAS_SHORT sensor, ret_val, accum_bin_stat, *sen_numbers, num_sen;
    SDDAS_SHORT *num_units, data_block, uind_raw, sen_units;
    SDDAS_SHORT num_bands, num_converted, fill_code, btime_yr;
    SDDAS_SHORT bttime_day, etime_day, start_time_yr, *start_time_day;
    SDDAS_SHORT *end_time_yr, *end_time_day, spin_sen_ctrl;
    SDDAS_CHAR extension[3], data_type, hdr_change, num_tbls, err_str[200];
    SDDAS_CHAR *ret_bin, *bin_stat, *base_bin;
    SDDAS_CHAR tbls_to_apply[5], num_center_band;
    short hr, min, sec;
    char more_data = 1, first_time = 1;
    void *idf_data_ptr;

    btime_yr = 2002;
    btime_day = 301;
    btime_sec = (19 * 3600) + (30 * 60) + 0;
    btime_nsec = 0;
    etime_yr = 2002;
    etime_day = 301;
    etime_sec = (19 * 3600) + (30 * 60) + 20;
    etime_nsec = 0;
```
strcpy (extension,"");
CfgInit();
dbInitialize();
init_idfs();

/**************************************************
/* Retreive the key that is associated with the project, mission, experiment, instrument */
/* and virtual instrument specified.                        */
/**************************************************/

ret_val = get_data_key ("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH",
                        &data_key);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from get_data_key routine.\n", ret_val);
    exit (-1);
}
get_version_number (&vnum);

/**************************************************
/* Create one instance of the data structure.             */
/**************************************************/

ret_val = create_idf_data_structure (&idf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from create_idf_data_structure routine.\n", ret_val);
    exit (-1);
}

/**************************************************
/* Open the data files associated with the time period selected for this data set */
/* extension / version combination.                    */
/**************************************************/

ret_val = file_open (data_key, extension, vnum, btime_yr, btime_day, btime_sec,
                     btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec, 0);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from file_open routine.\n", ret_val);
    exit (-1);
}
/** Find the position in the data file closest to the requested start time for this data set. */
/** Unlike real-time, if an error is encountered, the system should terminate - no need */
/** to retry in anticipation of incoming data. */
***********************************************************************/

ret_val = file_pos (data_key, extension, vnum, idf_data_ptr, btime_yr, btime_day,
                   btime_sec, btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec);
if (ret_val != ALL_OKAY)
{
  printf ("\n Error %d from file_pos routine.\n", ret_val);
  exit (-1);
}

spin_sen_ctrl = 0; /* sensor 0 as time controller */
ret_val = start_of_spin (data_key, extension, vnum, spin_sen_ctrl, etime_yr, etime_day,
                         etime_sec, etime_nsec);
if (ret_val != ALL_OKAY)
{
  printf ("\n Error %d from start_of_spin routine.\n", ret_val);
  exit (-1);
}

/***************************************************************************/
/* Get the data for all sensors. */
/***************************************************************************/

EXP_DATA = (struct idf_data *) idf_data_ptr;
while (more_data)
{
  /***************************************************************************/
  /* Some items need to be set just once. */
  /***************************************************************************/

  if (first_time)
  {
    /***************************************************************************/
    /* Select sensor data for sensor zero. */
    /***************************************************************************/

    first_time = 0;
    num_tbls = 0;
    tbls_to_apply[0] = 1;
    tbl_oper[0] = 0;
    data_type = SENSOR;
  }
sen_min = VALID_MIN;
sen_max = VALID_MAX;
sensor = spin_sen_ctrl;
ret_val = fill_sensor_info (data_key, extension, vnum, sensor, sen_min, sen_max, num_tbls,
tbls_to_apply, tbl_oper, data_type, 0);
if (ret_val != ALL_OKAY)
{
  printf ("\n Error %d from fill_sensor_info routine.\n", ret_val);
  exit (-1);
}

/******************************************************************************/
/*  Create the data bins.                                                   */
/******************************************************************************/

num_center_band = 0;
ret_val = set_bin_info (data_key, extension, vnum, VARIABLE_SWEEP, 0.0, 93.0, 1.0,
93, LIN_SPACING, num_center_band, tbls_to_apply, tbl_oper,
num_center_band, tbls_to_apply, tbl_oper,
num_center_band, tbls_to_apply, tbl_oper,
'L', POINT_STORAGE, NO_BIN_FILL);
if (ret_val != ALL_OKAY)
{
  printf ("\n Error %d from set_bin_info routine.\n", ret_val);
  exit (-1);
}

/******************************************************************************/
/*  The scan units should be in terms of electron volts.                    */
/******************************************************************************/

num_tbls = 0;
tbls_to_apply[0] = 0;
tbl_oper[0] = 0;
ret_val = set_scan_info (data_key, extension, vnum, num_tbls, tbls_to_apply, tbl_oper);
if (ret_val != ALL_OKAY)
{
  printf ("\n Error %d from set_scan_info routine.\n", ret_val);
  exit (-1);
}

/******************************************************************************/
/*  Since all sensors use the same scan range, any valid sensor number can be */
/*  passed in as the required argument.                                      */
/******************************************************************************/
ret_val = center_and_band_values (data_key, extension, vnum, idf_data_ptr, sensor, 1, 1,
    &center_ptr, &band_low, &band_high, &num_bands,
    &num_converted);
if (ret_val != ALL_OKAY && ret_val != CENTER_CONVERSION)
    {
        printf ("\n Error %d from center_and_band_values routine.\n", ret_val);
        exit (-1);
    }
}

fill_code = spin_data (data_key, extension, vnum, idf_data_ptr, &sen_numbers,
    &ret_data, &ret_frac, &ret_bin, &num_sen, &num_units,
    &data_block, &start_time_yr, &start_time_day, &start_time_sec,
    &start_time_nano, &end_time_yr, &end_time_day,
    &end_time_sec, &end_time_nano, &hdr_change);
if (fill_code != ALL_OKAY)
    {
        printf ("\n Error %d from spin_data routine.\n", ret_val);
        exit (-1);
    }

/**************************************************************************/
/*  Loop over sensors processed by the SPIN_DATA routine.                        */
/**************************************************************************/
for (i = 0; i < num_sen; ++i)
    {
        if (*(sen_numbers + i) == sensor)
            {
                offset_unit = *(num_units + i) * data_block;
                base_data = ret_data + offset_unit;
                base_frac = ret_frac + offset_unit;
                base_bin = ret_bin + offset_unit;

                num_tbls = 0;
                tbls_to_apply[0] = 1;
                tbl_oper[0] = 0;
                ret_val = units_index (data_key, extension, vnum, sensor, sen_min, sen_max, tbls_to_apply,
                    tbl_oper, data_type, 0, &uind_raw, &sen_units, num_tbls);
                if (ret_val != ALL_OKAY)
                    {
                        printf ("\n Error %d from units_index routine.\n", ret_val);
                        exit (-1);
                    }
            }
if (fill_code == ALL_OKAY)
{
    /*************************************************/
    /*  Print the raw units data.          */
    /*************************************************/
    offset_unit = uind_raw * data_block;
    dptr = base_data + offset_unit;
    frac = base_frac + offset_unit;
    bin_stat = base_bin + offset_unit;
    accum_bin_stat = 0;
    
    for (loop = 0; loop < data_block; ++loop)
    {
        accum_bin_stat += *(bin_stat + loop);
    }
    /*  Make sure there is some data in the buffers. Still check   */
    /*  bin_stat since frac will be 0.0 if bin_stat = 0. For this    */
    /*  data set, it is known that the sweep values are contiguous. */
    if (accum_bin_stat != 0)
    {
        frac_sec = *start_time_nano / 1000000000.0;
        printf("nTIME %04d %03d ", *start_time_yr, *start_time_day);
        hr = *start_time_sec / 3600;
        sec = *start_time_sec % 3600;
        min = sec / 60;
        sec = sec % 60;
        printf("%02d %02d %02d %9.6f", hr, min, sec, frac_sec);
        for (loop = 0; loop < data_block; ++loop)
        {
            if (*(bin_stat + loop) != 0)
            {
                printf("n%8.3f %3.0f", *(dptr+loop) / *(frac+loop), (float) loop);
            }
            else
            {
                printf("n%10.2e %3.0f", *(dptr + loop), (float) loop);
            }
        }
    }
}
if (*start_time_yr == etime_yr && *start_time_day == etime_day &&
  *start_time_sec > etime_sec)
  more_data = 0;
else if (*start_time_yr == etime_yr && *start_time_day == etime_day &&
  *start_time_sec == etime_sec && *start_time_nano > etime_nsec)
  more_data = 0;
}
free_experiment_info();
}
EXAMPLE 7

This example demonstrates how a programmer would go about developing a program that utilizes the SCF software. The data that is returned is displayed to the screen, but these values could be sent to a file to be used by another program or some other action could be taken once the results were retrieved.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "SCF_defs.h"
#include "SCF_file_defs.h"
#include "libbase_SCF.h"
#include "libCfg.h"
#include "libdb.h"

void main (int argc, char **argv)
{
    struct scf_data *SCF_DATA;
    register SDDAS_LONG i;
    SDDAS_FLOAT *dptr, *stop_loop;
    SDDAS_DOUBLE time_value;
    SDDAS_LONG btime_sec, etime_sec, ret_time_sec, ret_time_nano;
    SDDAS_LONG etime_nano, btime_nano, num_output, *intptr, *dimen;
    SDDAS_USHORT scf_vnum;
    SDDAS_SHORT ret_val, btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_CHAR filename[SCF_FILENAME];
    char more_data = 1;
    void *scf_data_ptr;

    /**********************************************/
    /*  Set the time for processing.               */
    /**********************************************/
    btime_yr = 1992;
    btime_day = 217;
    btime_sec = 32340;
    btime_nano = 0;

    etime_yr = 1992;
    etime_day = 217;
    etime_sec = 32342;
    etime_nano = 0;

    strcpy (filename, "TMMO_EXAMPLE");
```
CfgInit();
dblInitialize();
initscf();

/*********************
/* Open the SCF file and all input data sets. */
**********************/

scf_version_number(&scf_vnum);
ret_val = scf_open(filename, scf_vnum, btime_yr, btime_day, btime_sec, btime_nano,
etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf("\n Error %d from scf_open routine.\n", ret_val);
    exit(-1);
}

/*********************
/* Position the input data sets at the requested start time. */
**********************/

ret_val = scf_position(filename, scf_vnum, btime_yr, btime_day, btime_sec,
                        btime_nano, etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf("\n Error %d from scf_position routine.\n", ret_val);
    exit(-1);
}

/*********************
/* Create one instance of the data structure. */
**********************/

ret_val = create_scf_data_structure(filename, scf_vnum, &scf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf("\n Error %d from create_scf_data_structure routine.\n", ret_val);
    exit(-1);
}
SCF_DATA = (struct scf_data *) scf_data_ptr;

/*********************
/* Find the input source with the fastest sample rate. */
**********************/
time_value = 0.0;

ret_val = scf_sample_rate (filename, scf_vnum, SCF_DELTA_T, time_value, SCF_MEASURE_LAT_TM);
if (ret_val != ALL_OKAY)
{
    printf ("n Error %d from scf_sample_rate routine.
", ret_val);
    exit (-1);
}

/**************************************************************/
/*  Determine the number of output variables being returned.  */
/**************************************************************/

ret_val = read_scf (filename, scf_vnum, S_NUM_OUTPUT, NOT_USED, (SDDAS_CHAR *) &num_output);
if (ret_val != ALL_OKAY)
{
    printf ("n Error %d from read_scf routine.
", ret_val);
    exit (-1);
}

intptr = (SDDAS_LONG *) calloc ((size_t) num_output, sizeof (SDDAS_LONG));
if (intptr == 0)
{
    printf ("n Error from calloc system call.
");
    exit (-1);
}

/**************************************************************/
/* Retrieve the "dimensionality" of the output variables. */
/**************************************************************/

for (dimen = intptr, i = 0; i < num_output; ++i, ++dimen)
{
    ret_val = read_scf (filename, scf_vnum, S_OUTPUT_DIMENSION, i, (SDDAS_CHAR *) dimen);
    if (ret_val != ALL_OKAY)
    {
        printf ("n Error %d from read_scf routine.
", ret_val);
        exit (-1);
    }
}
dimen = intptr;
while (more_data)
{
    ret_val = scf_output_data (filename, scf_vnum, scf_data_ptr);
    if (ret_val != ALL_OKAY && ret_val != SCF_TERMINATE)
    {
        printf ("Error %d from scf_output_data routine.\n", ret_val);
        exit (-1);
    }

    printf ("START YEAR = %d START DAY = %d", SCF_DATA->byear,
            SCF_DATA->bday);
    printf ("START TIME_MS = %ld START TIME_NS = %ld", SCF_DATA->bmilli,
            SCF_DATA->bnano);
    printf ("END YEAR = %d END DAY = %d", SCF_DATA->eyear,
            SCF_DATA->eday);
    printf ("END TIME_MS = %ld END TIME_NS = %ld", SCF_DATA->emilli,
            SCF_DATA->enano);

    for (i = 0; i < SCF_DATA->num_output; ++i)
    {
        dptr = SCF_DATA->output_data + *(SCF_DATA->output_index + i);
        stop_loop = dptr + *(SCF_DATA->output_length + i);

        if (*(dimen + i) == 0)
            for (; dptr < stop_loop; ++dptr)
                printf ("Output Variable %ld = %e", i, *dptr);
    }
}
if (ret_val == SCF_TERMINATE)
{
    more_data = 0;
    break;
}

ret_time_sec = (SCF_DATA->emilli + (SCF_DATA->enano / 1000000)) / 1000;
ret_time_nano = (SCF_DATA->emilli % 1000) * 1000000 + SCF_DATA->enano;

if (SCF_DATA->eyear > etime_yr ||
    (SCF_DATA->eyear == etime_yr && SCF_DATA->eday > etime_day) ||
    (SCF_DATA->eyear == etime_yr && SCF_DATA->eday == etime_day &&
        ret_time_sec > etime_sec) ||
    (SCF_DATA->eyear == etime_yr && SCF_DATA->eday == etime_day &&
        ret_time_sec == etime_sec && ret_time_nano > etime_nano))
{
    more_data = 0;
    break;
}
}

free_scf_info ();
exit (0);
EXAMPLE 8

This example demonstrates how a programmer would go about developing a program that utilizes the SCF software to return time-averaged data.

```c
#include <stdio.h>
#include <string.h>
#include "SCF_defs.h"
#include "user_defs.h"
#include "libbase_SCF.h"
#include "libavg_SCF.h"
#include "libCfg.h"
#include "libdb.h"

void main (int argc, char **argv)
{
    struct scf_data *SCF_DATA;
    register SDDAS_LONG i, loop, buf_num;
    register SDDAS_FLOAT *dptr, *frac;
    SDDAS_DOUBLE frac_sec;
    SDDAS_FLOAT *ret_data, *ret_frac, *base_data, *base_frac;
    SDDAS_FLOAT time_frac, data_min, data_max, *center_bin, *bin_low, *bin_high;
    SDDAS_ULONG buf_zero_loc;
    SDDAS_LONG stime_sec, stime_nano, end_time_sec, end_time_nano, offset_unit;
    SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano, offset_buf;
    SDDAS_LONG base_sec, base_nano, res_sec, res_nano, base_pix, output_var;
    SDDAS_LONG dependent_var, accum_bin_stat, block_size, *bpix, *epix;
    SDDAS_LONG num_output, *output_numbers, *output_size, num_select, output_ind;
    SDDAS_LONG time_msec, num_bands;
    SDDAS_USHORT scf_vnum;
    SDDAS_SHORT ret_val, btime_yr, btime_day, etime_yr, etime_day, end_time_yr , ret_code;
    SDDAS_SHORT base_yr, base_day, hr, min, sec, stime_yr, stime_day, end_time_day ;
    SDDAS_CHAR filename[SCF_FILENAME], *ret_bin, *base_bin;
    SDDAS_CHAR *buf_stat, *bin_stat;
    char more_data = 1;
    void *scf_data_ptr;

    /***************************************************************************/
    /*  Set the time for processing.                                      */
    /***************************************************************************/

    btime_yr = 1992;
    btime_day = 217;
    btime_sec = 32340;
    btime_nano = 0;
}
```
etime_yr = 1992;
etime_day = 217;
etime_sec = 32342;
etime_nano = 0;

strcpy (filename, "TMMO_EXAMPLE");
CfgInit();
dbInitialize();
init_scf();

/**********************************************************/
/* Open the SCF file and all input data sets. */
/**********************************************************/

scf_version_number (&scf_vnum);
ret_val = scf_open (filename, scf_vnum, btime_yr, btime_day, btime_sec, btimeNano, 
etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf ("%d Error %d from scf_open routine.\n", ret_val);
    exit (-1);
}

/**************************************************************/
/* Position the input data sets at the requested start time. */
/**************************************************************/

ret_val = scf_position (filename, scf_vnum, btime_yr, btime_day, btime_sec, 
                        btime_nano, etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf ("%d Error %d from scf_position routine.\n", ret_val);
    exit (-1);
}

/**************************************************************/
/* Create one instance of the data structure. */
/**************************************************************/

ret_val = create_scf_data_structure (filename, scf_vnum, &scf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf ("%d Error %d from create_scf_data_structure routine.\n", ret_val);
    exit (-1);
}
SCF_DATA = (struct scf_data *) scf_data_ptr;

/************************************************************************/
/*  Find the input source with the fastest sample rate.                  */
/************************************************************************/

time_frac = 0.0;
ret_val = scf_sample_rate (filename, scf_vnum, SCF_DELTA_T, time_frac,
   SCF_MEASURE_LAT_TM);
if (ret_val != ALL_OKAY)
{
   printf ("\n Error %d from scf_sample_rate routine.\n", ret_val);
   exit (-1);
}

/************************************************************************/
/*  Set the base reference time, location and duration for the data buffers.            */
/************************************************************************/

ret_val = scf_algorithm_start (filename, scf_vnum, &base_yr, &base_day,
   &base_sec, &base_nano, &res_sec, &res_nano);
if (ret_val < 0)
{
   printf ("\n Error %d from scf_algorithm_start routine.\n", ret_val);
   exit (-1);
}

base_pix = 0;
scf_time_reference (scf_vnum, base_yr, base_day, base_sec, base_nano,
   base_pix, res_sec, res_nano);

/************************************************************************/
/*  Create the data bins for output variable zero.            */
/************************************************************************/

output_var = 0;
dependent_var = -1;
ret_val = scf_bin_info (filename, scf_vnum, output_var, FIXED_SWEEP,
   0.0, 0.0, 1, LIN_SPACING, dependent_var,
   dependent_var, dependent_var, ', ', POINT_STORAGE);
if (ret_val != ALL_OKAY)
{
   printf ("\n Error %d from scf_bin_info routine.\n", ret_val);
   exit (-1);
}
/*********************************************************************/
/*  Select output variable zero, which is known to be scalar, so no dependent */
/*  variable needed.                                      */
/*********************************************************************/
data_min = VALID_MIN;
data_max = VALID_MAX;
ret_val = scf_output_select (filename, scf_vnum, output_var, data_min, data_max,
dependent_var);
if (ret_val != ALL_OKAY)
{
    printf ("\\n Error %d from scf_output_select routine.\\n", ret_val);
    exit (-1);
}
/**************************************************************************/
/*  Evaluate the algorithm, retrieving the output until the requested end time is reached.   */
/**************************************************************************/
while (more_data)
{
    ret_code = scf_time_average (filename, scf_vnum, scf_data_ptr, &ret_data, &ret_frac,
        &ret_bin, &bpix, &epix, &buf_stat, &stime_yr, &stime_day,
        &stime_sec, &stime_nano, &end_time_yr, &end_time_day,
        &end_time_sec, &end_time_nano, &num_output,
        &output_numbers, &output_size);
    if (ret_code != ALL_OKAY && ret_code != SCF_TERMINATE)
    {
        printf ("\\n Error %d from scf_time_average routine.\\n", ret_code);
        exit (-1);
    }
    /**************************************************************************/
    /*  Loop over output variables processed by scf_time_average().    */
    /**************************************************************************/
    for (i = 0; i < num_output; ++i)
    {
        if (*((output_numbers + i)) == output_var)
        {
            ret_val = scf_output_center_and_bands (filename, scf_vnum, output_var,
                &center_bin, &bin_low, &bin_high, &num_bands);
            if (ret_val != ALL_OKAY)
            {
                printf ("\\n Error %d from scf_output_center_and_bands routine.\\n", ret_val);
            }
        }
    }
```c
exit (-1);
}
ret_val = scf_output_data_index (filename, scf_vnum, output_var, data_min, data_max,
    dependent_var, &num_select, &output_ind, &buf_zero_loc);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from scf_output_data_index routine.\n", ret_val);
    exit (-1);
}

/******************************************************************
/*  Set pointers to the beginning of the first buffer for the selected output variable.*/
/******************************************************************/

base_data = ret_data + buf_zero_loc;
base_frac = ret_frac + buf_zero_loc;
base_bin = ret_bin + buf_zero_loc;
block_size = *(output_size + i);

for (buf_num = 0; buf_num < NUM_BUFFERS; ++buf_num)
if (*(*(buf_stat + buf_num) == BUFFER_READY)
{
    /********************************************************************************/
    /*  Point to the buffer being processed, then point to the definition         */
    /*  being processed.                           */
    /********************************************************************************/

    offset_buf = buf_num * num_select * block_size;
    offset_unit = output_ind * block_size;

    dptr = base_data + offset_buf + offset_unit;
    frac = base_frac + offset_buf + offset_unit;
    bin_stat = base_bin + offset_buf + offset_unit;
    accum_bin_stat = 0;

    for (loop = 0; loop < block_size; ++loop)
        accum_bin_stat += *(bin_stat + loop);

    /********************************************************************************/
    /*  Make sure there is some data in the buffers. For this data, the bins are */
    /*  contiguous since FIXED_SWIPE LIN_SPACING was utilized for */
    /*  scf_bin_info().                                   */
    /********************************************************************************/

```
if (accum_bin_stat != 0)
{
    frac_sec = stime_nano / 1000000000.0;
    hr = stime_sec / 3600;
    sec = stime_sec % 3600;
    min = sec / 60;
    sec = sec % 60;
    time_msec = stime_sec * 1000 + (stime_nano / 1000000);
    printf ("TIME %04d%03d ", stime_yr, stime_day);
    printf ("%02d %02d %02d %9.6f", hr, min, sec, frac_sec);

    for (loop = 0; loop < block_size; ++loop)
    {
        if (*(bin_stat + loop) != 0)
            printf ("%e bin_low = %e bin_high = %e", *(dptr+loop) / *(frac+loop),
                    *(bin_low + loop), *(bin_low + loop + 1));
        else
            printf ("%10.2e bin_low = %e bin_high = %e", *(dptr + loop),
                    *(bin_low + loop), *(bin_low + loop + 1));
    }
}

/**************************************************************************/
/*  Processing must terminate due to data not being on-line.            */
/**************************************************************************/
if (ret_code == SCF_TERMINATE)
    break;

/**************************************************************************/
/*  End time has been reached? Compare against the end time of the iteration since */
/*  requested end time could fall between the time range processed.        */
/**************************************************************************/
if (end_time_yr == etime_yr && end_time_day == etime_day && end_time_sec > etime_sec)
    break;
else if (end_time_yr == etime_yr && end_time_day == etime_day &&
    end_time_sec == etime_sec && end_time_nano > etime_nano)
    break;
}
free_scf_info ();
exit (0);
EXAMPLE 9

This example demonstrates how a programmer would go about developing a program that utilizes the SCF software to return sample-averaged data.

```c
#include <stdio.h>
#include <string.h>
#include "SCF_defs.h"
#include "user_defs.h"
#include "libbase_SCF.h"
#include "libavg_SCF.h"
#include "libCfg.h"
#include "libdb.h"

void main (int argc, char **argv)
{
    struct scf_data *SCF_DATA;
    register SDDAS_LONG i, loop;
    register SDDAS_FLOAT *dptr, *frac;
    SDDAS_DOUBLE frac_sec;
    SDDAS_FLOAT *ret_data, *ret_frac, *base_data, *base_frac;
    SDDAS_FLOAT time_frac, data_min, data_max, *center_bin, *bin_low, *bin_high;
    SDDAS_ULONG buf_zero_loc;
    SDDAS_LONG stime_sec, stime_nano, end_time_sec, end_time_nano, offset_unit;
    SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano, time_msec, num_bands;
    SDDAS_LONG dependent_var, accum_bin_stat, block_size, output_var;
    SDDAS_LONG num_output, *output_numbers, *output_size, num_select, output_ind;
    SDDAS_USHORT scf_vnum;
    SDDAS_SHORT ret_code, ret_val, btime_yr, btime_day, etime_yr, etime_day;
    SDDAS_SHORT hr, min, sec, stime_yr, stime_day, end_time_yr, end_time_day;
    SDDAS_CHAR filename[SCF_FILENAME], *ret_bin, *base_bin, *bin_stat;
    char more_data = 1;
    void *scf_data_ptr;

    /**********************************************************************************************/
    /*  Set the time for processing.                                                           */
    /**********************************************************************************************/

    btime_yr = 1992;
    btime_day = 217;
    btime_sec = 32340;
    btime_nano = 0;

    etime_yr = 1992;
    etime_day = 217;
```

etime_sec = 32342;
etime_nano = 0;
strcpy (filename, "TMMO_EXAMPLE");
CfgInit ();
dbInitialize ();
init_scf ();

/**************************************************************************/
/*  Open the SCF file and all input data sets.                  */
/**************************************************************************/

scf_version_number (&scf_vnum);
ret_val = scf_open (filename, scf_vnum, btime_yr, btime_day, btime_sec, btime_nano,
etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from scf_open routine.\n", ret_val);
    exit (-1);
}

/*******************************************************************************/
/*  Position the input data sets at the requested start time.          */
/*******************************************************************************/

ret_val = scf_position (filename, scf_vnum, btime_yr, btime_day, btime_sec,
                        btime_nano, etime_yr, etime_day, etime_sec, etime_nano);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from scf_position routine.\n", ret_val);
    exit (-1);
}

/******************************************************************************/
/*  Create one instance of the data structure.                                */
/******************************************************************************/

ret_val = create_scf_data_structure (filename, scf_vnum, &scf_data_ptr);
if (ret_val != ALL_OKAY)
{
    printf ("\n Error %d from create_scf_data_structure routine.\n", ret_val);
    exit (-1);
}

SCF_DATA = (struct scf_data *) scf_data_ptr;
/* Find the input source with the fastest sample rate. */

time_frac = 0.0;
ret_val = scf_sample_rate (filename, scf_vnum, SCF_DELTA_T, time_frac,
SCF_MEASURE_LAT_TM);
if (ret_val != ALL_OKAY)
{
  printf ("n Error %d from scf_sample_rate routine.\n", ret_val);
  exit (-1);
}

/* Create the data bins for output variable zero. */

output_var = 0;
dependent_var = -1;
ret_val = scf_bin_info (filename, scf_vnum, output_var, FIXED_SWEEP,
  0.0, 0.0, 1, LIN_SPACING, dependent_var,
  dependent_var, dependent_var, ',', POINT_STORAGE);
if (ret_val != ALL_OKAY)
{
  printf ("n Error %d from scf_bin_info routine.\n", ret_val);
  exit (-1);
}

/* Select output variable zero, which is known to be scalar, so no dependent */
/* variable needed. */

data_min = VALID_MIN;
data_max = VALID_MAX;
ret_val = scf_output_select (filename, scf_vnum, output_var, data_min, data_max,
dependent_var);
if (ret_val != ALL_OKAY)
{
  printf ("n Error %d from scf_output_select routine.\n", ret_val);
  exit (-1);
}

/* Evaluate the algorithm, averaging two iterations of the algorithm until the requested */
/* end time has been reached. */
while (more_data)
{
  ret_code = scf_sample_average (filename, scf_vnum, scf_data_ptr, 2, &ret_data, &ret_frac,
  &ret_bin, &stime_yr, &stime_day, &stime_sec, &stime_nano,
  &end_time_yr, &end_time_day, &end_time_sec, &end_time_nano,
  &num_output, &output_numbers, &output_size);
  if (ret_code != ALL_OKAY && ret_code != SCF_TERMINATE)
  {
    printf ("\n Error %d from scf_sample_average routine.\n", ret_code);
    exit (-1);
  }
}

/******************************************************************************/
/*  Loop over output variables processed by scf_sample_average().            */
/******************************************************************************/

for (i = 0; i < num_output; ++i)
{
  if (*(output_numbers + i) == output_var)
  {
    ret_val = scf_output_center_and_bands (filename, scf_vnum, output_var,
      &center_bin, &bin_low, &bin_high, &num_bands);
    if (ret_val != ALL_OKAY)
    {
      printf ("\n Error %d from scf_output_center_and_bands routine.\n", ret_val);
      exit (-1);
    }
    ret_val = scf_output_data_index (filename, scf_vnum, output_var, data_min, data_max,
      dependent_var, &num_select, &output_ind,
      &buf_zero_loc);
    if (ret_val != ALL_OKAY)
    {
      printf ("\n Error %d from scf_output_data_index routine.\n", ret_val);
      exit (-1);
    }
  }
  
  /******************************************************************************/
  /*  Set pointers to the beginning of the data for the selected output variable.     */
  /******************************************************************************/

  base_data = ret_data + buf_zero_loc;
  base_frac = ret_frac + buf_zero_loc;
  base_bin = ret_bin + buf_zero_loc;
  block_size = *(output_size + i);
/*******************************/
/*  Point to the definition being processed. */
/*******************************/

offset_unit = output_ind * block_size;
dptr = base_data + offset_unit;
frac = base_frac + offset_unit;
bin_stat = base_bin + offset_unit;
accum_bin_stat = 0;

for (loop = 0; loop < block_size; ++loop)
  accum_bin_stat += *(bin_stat + loop);

/*******************************/
/*  Make sure there is some data in the buffer. */
/*******************************/

if (accum_bin_stat != 0)
{
  frac_sec = stime_nano / 1000000000.0;
  hr = stime_sec / 3600;
  sec = stime_sec % 3600;
  min = sec / 60;
  sec = sec % 60;
  printf ("TIME %04d %03d ", stime_yr, stime_day);
  printf ("%02d %02d %02d %9.6f", hr, min, sec, frac_sec);

  for (loop = 0; loop < block_size; ++loop)
  {
    if (*((bin_stat + loop) != 0)
        printf ("%e", *(dptr+loop) / *(frac+loop));
    else
        printf ("%10.2e", *(dptr + loop));
  }
}
}

/*******************************/
/*  Processing must terminate due to data not being on-line. */
/*******************************/

if (ret_code == SCF_TERMINATE)
  break;
/************************************************************/ 
/* End time has been reached? Compare against the end time of the iteration since */ 
/* requested end time could fall between the time range processed. */ 
/************************************************************/ 

if (end_time_yr == etime_yr && end_time_day == etime_day && end_time_sec > etime_sec) 
    break;
else if (end_time_yr == etime_yr && end_time_day == etime_day &&
    end_time_sec == etime_sec && end_time_nano > etime_nano) 
    break;
}

free_scf_info();
exit(0);
ADJUST_TIME
  function - adjusts the time components if a year/day boundary has been crossed

SYNOPSIS
  #include "libbase_idfs.h"

  void adjust_time (SDDAS_SHORT *year, SDDAS_SHORT *day, SDDAS_LONG *time,
                   SDDAS_CHAR time_unit)

ARGUMENTS
  year  - the year time component
  day   - the day of year time component
  time  - the time of day time component
  time_unit - flag which specifies the time unit for the time argument
              1 – the time of day component is specified in seconds
              2 – the time of day component is specified in milliseconds

DESCRIPTION
  Adjust_time is the IDFS routine that will correct time components when a day boundary or
  boundaries have been crossed. Year boundaries and leap years are taken into account
  during the calculation. If the time values are correct as is, the time values are not modified
  in any way.

ERRORS
  This routine returns no status or error codes.

BUGS
  None

EXAMPLES
  Correct the time values in case the time of day value represents more than a day's worth of
  milliseconds. Assume that time values have already been set.

  #include "libbase_idfs.h"

  SDDAS_SHORT year, day;
  SDDAS_LONG tod_milli;
  SDDAS_CHAR time_unit;

  time_unit = 2;
  adjust_time (&year, &day, &tod_milli, time_unit);
CALC_TIME_RESOLUTION

function - returns the maximum temporal resolution allowed by the selected data set

SYNOPSIS

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT calc_time_resolution (SDDAS_ULONG data_key,
                                  SDDAS_CHAR *exten, SDDAS_USHORT version,
                                  void *idf_data_ptr, SDDAS_SHORT num_sweeps,
                                  SDDAS_LONG *res_sec, SDDAS_LONG *res_nsec)
```

ARGUMENTS

- `data_key` - unique value which indicates the data set of interest
- `exten` - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- `version` - IDFS data set identification number which allows for multiple openings of the same data set
- `idf_data_ptr` - pointer to the `idf_data` structure that is to hold sensor data and pertinent ancillary data for the data set of interest
- `num_sweeps` - the number of sweeps / spins to use to calculate the maximum temporal resolution
  - negative value indicates spins are to be utilized for the calculation
  - positive value indicates sweeps are to be utilized for the calculation
- `res_sec` - the temporal resolution expressed in seconds
- `res_nsec` - the temporal resolution residual of seconds expressed in nanoseconds
- `calc_time_resolution` - routine status (see TABLE 1)

**TABLE 1.** Status Codes Returned for `CALC_TIME_RESOLUTION`

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALC_TRES_NOT_FOUND</td>
<td>the requested <code>data_key</code>, <code>exten</code>, <code>version</code> combination has no memory allocated for processing (user did not call <code>file_open</code> for this combination)</td>
</tr>
<tr>
<td>WRONG_HEADER_FORMAT</td>
<td>multi-dimensional IDFS data storage is not supported by this module</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

`calc_time_resolution` is the IDFS routine that determines the maximum temporal resolution associated with the data set of interest, which is referenced through the key value `data_key` which can be created using the `get_data_key` module. The calculated resolution may be modified by specifying the number of sweeps/spins which are to be processed. If the parameter `num_sweeps` is set to indicate that the resolution should be calculated in terms of spins, a check is made to determine if the data set requested returns a spin rate. If
the data set does not return a spin rate, the resolution is calculated in terms of sweeps, not
spins. Before this routine can be utilized, a call to the routine `file_open` must be made.

The parameter `version` allows multiple file openings for an IDFS data set. If the data,
header and VIDF file for the specified data set need to be opened just once for processing,
the same version number should be passed to all IDFS routines. However, for multiple file
openings, the version number should be unique and all file manipulations performed by the
IDFS routines will use the file descriptors defined for the version number specified. The
user should call the `get_version_number` routine to retrieve a unique version number
instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single
version number will suffice.

The parameter `idf_data_ptr` is a pointer to the structure that is to hold all data pertinent to
the data set being processed. The structure is created and the address to this structure is
returned when a call to the `create_idf_data_structure` routine is made. The user also has
the option of calling the module `create_data_structure`, which determines what type of
data structure is needed for the IDFS data set of interest. In most cases, one data structure is
sufficient to process any number of distinct data sets. However, if more than one structure
is needed, the user may call the `create_idf_data_structure` routine N times to create N
instances of the `idf_data` structure. The user must keep track of which pointer to send to
the IDFS routines that utilize this structure.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of
IDFS files, the two character extension applied to these data files must be supplied to this
routine within the string variable `exen`. These files must have the identical name as the
IDFS files with the two character identification code appended to the end of the file names
(i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files
must reside either in the directory specified by the environment variable USER_DATA,
which is set by the user, or in the user's home directory if the environment variable
USER_DATA is not set. To open the default IDFS data files, `exen` should be set to a null
string. The usage of modified data sets is limited to post acquisition data; therefore, it is
suggested that the user set `exen` to a null string for real-time scenarios.

**ERRORS**
All errors within this routine are returned through the status variable. The include file
`ret_codes.h`, which includes all possible return values, should be included so that the
mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in
section 1H of the IDFS Programmers Manual.

**SEE ALSO**
- `file_open` 1R
- `get_data_key` 1R
- `get_version_number` 1R
- `create_data_structure` 1R
EXAMPLES

Determine the temporal resolution for four sweeps of data from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_LONG res_sec, res_nsec;
SDDAS_SHORT status;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = calc_time_resolution (data_key, ",", vnum, idf_data_ptr, 4,
                         &res_sec, &res_nsec);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by calc_time_resolution routine.\n", status);
    exit (-1);
}
```
CONVERT_TO_UNITS

function - converts raw data into units by applying the tables and operations specified

SYNOPSIS

#include "libbase_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT convert_to_units (SDDAS_ULONG data_key, SDDAS_CHAR *exten, 
    SDDAS_USHORT version, void *idf_data_ptr, 
    SDDAS_SHORT sensor, SDDAS_CHAR data_type, 
    SDDAS_CHAR cal_set, SDDAS_CHAR num_tbls, 
    SDDAS_CHAR *tbls_to_apply, SDDAS_LONG *tbl_oper, 
    SDDAS_FLOAT *ret_data, SDDAS_CHAR chk_fill, 
    SDDAS_LONG fill_value)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple openings of the same data set
idf_data_ptr - pointer to the idf_data structure that is to hold sensor data and pertinent ancillary data for the data set of interest
sensor - the sensor identification number if processing sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, or spacecraft potential data; otherwise, the instrument status (mode) value of interest
data_type - the type of data being requested
    1 - sensor data (SENSOR)  
    2 - sweep step data (SWEEP_STEP)  
    3 - calibration data (CAL_DATA)  
    4 - instrument status or mode data (MODE)  
    5 - data quality data (D_QUAL)  
    6 - pitch angle data (PITCH_ANGLE)  
    7 - start azimuthal angle data (START_AZ_ANGLE)  
    8 - stop azimuthal angle data (STOP_AZ_ANGLE)  
    9 – spacecraft potential data (SC_POTENTIAL)  
    10 – background data (BACKGROUND)
cal_set - the calibration set from which requested calibration data (CAL_DATA) is to be retrieved
    - If calibration data is not being requested, this parameter is not utilized and it is suggested that the user pass a value of zero for this parameter.
num_tbls - the number of elements specified in the tbls_to_apply and tbl_oper parameters
tbls_to_apply - the tables that are to be applied in order to derive the desired units
tbl_oper - the operations that are to be applied to the specified tables in order to derive the desired units
ret_data - user-defined array which holds the data in the unit requested
chk_fill - flag indicating if the data is to be checked for fill values. If a fill value is found within the data and if the chk_fill flag is set to 1, the data value will be returned as -3.4e38 (OUTSIDE_MIN) in the ret_data array in order to flag the data as a fill value. If the chk_fill flag is set to 0, all data is treated as valid data and is returned as such in the ret_data array. Fill data is only applicable to SENSOR and CAL_DATA data types.
fill_value - the fill data value, as specified within the raw telemetry
convert_to_units - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for CONVERT_TO_UNITS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNVT_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>CNVT_BAD_DTYPE</td>
<td>invalid data type value</td>
</tr>
<tr>
<td>CNVT_BAD_TBL_OPER</td>
<td>the look-up operation is not defined for the combination of the primary and intermediate accumulators</td>
</tr>
<tr>
<td>CNVT_BAD_TBL_NUM</td>
<td>invalid table number</td>
</tr>
<tr>
<td>CNVT_TMP</td>
<td>there is no data in the intermediate accumulator to combine with the primary accumulator</td>
</tr>
<tr>
<td>CNVT_TMP_MALLOC</td>
<td>no memory for the intermediate accumulator</td>
</tr>
<tr>
<td>CONV_CAL_MALLOC</td>
<td>no memory for temporary array that holds the calibration data before it is expanded to vector length</td>
</tr>
<tr>
<td>CONV_CAL_VECTOR_MISMATCH</td>
<td>one of the tables specified is a function of data that is not dimensioned the same size as the requested calibration set</td>
</tr>
<tr>
<td>CONV_MODE_BAD_MODE</td>
<td>invalid instrument status (mode) value</td>
</tr>
<tr>
<td>CONV_MODE_BAD_TBL_NUM</td>
<td>the table specified is not a mode-dependent table for the mode in question</td>
</tr>
<tr>
<td>CONV_MODE_MISMATCH</td>
<td>only tables that apply to mode data are valid</td>
</tr>
<tr>
<td>CNVT_NO_ADV</td>
<td>a table operator references an advanced data buffer(s) which does not contain any data to use in order to perform the specified operation</td>
</tr>
<tr>
<td>CNVT_BAD_BUF_NUM</td>
<td>a table operator specifies a combination function to be performed on advanced data buffers and specifies a 2 for the FROM or TO buffer value. A value of 2 is reserved and cannot be used.</td>
</tr>
<tr>
<td>CNVTSAME_BUF_NUM</td>
<td>a table operator specifies a combination function to be performed on advanced data buffers and specifies the same value for the FROM and TO buffer values – the two data buffers values must be different</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data being returned</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>
DESCRIPTION

Convert_to_units is the IDFS data conversion routine. This routine will convert sensor data, sweep step data, calibration data, instrument status (mode) information, data quality data, pitch angle data, azimuthal angle data, spacecraft potential data and background data to different formats (units) by applying the tables and the table operations in the specified order. It is imperative that a call to the read_drec routine be made PRIOR to calling this routine in order to fill the idf_data structure. With the exception of mode data, all data types are associated with a specific sensor, which is indicated through the sensor number (sensor). The sensor is further identified as being associated with a specific data set. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module.

There are two data types defined for azimuthal angle data, START_AZ_ANGLE and STOP_AZ_ANGLE. The start azimuthal angle values are always returned as values between 0 and 360 degrees. However, the stop azimuthal angle values could be negative (if the instrument is spinning in a negative direction) or could be greater than 360 degrees. The stop azimuthal angle values are computed by adding the degrees covered by the accumulation time of each sample to the start azimuthal angle values.

The instrument status (mode) data comes from the header record and is defined for the virtual instrument in question; therefore, the instrument status data is not associated with any particular sensor. When mode data is being requested, the user should set the sensor parameter to specify the status value of interest, with the numbering starting at zero.

The units that the data is to be returned in is specified by the user through the parameters num_tcb, tbls_to_apply and tbl_oper. If the user wants raw units, that is, the telemetry data, to be returned, the user should set the num_tcb parameter to zero and put a placeholder variable for the tbls_to_apply and tbl_oper parameters. The raw units for pitch angle and azimuthal angle data is defined as degrees. For other units pertinent to the data set requested, the user must specify the tables and the table operations that are to be applied to calculate the desired unit. The order is implied by the contents of the tbls_to_apply array.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter idf_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is
returned when a call to the `create_idf_data_structure` routine is made. The user also has the option of calling the module `create_data_structure`, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the `create_idf_data_structure` routine N times to create N instances of the `idf_data` structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

The user will need to pass in an array `ret_data` that will be used to return the data in the unit requested. The user has 2 choices as to how to handle the allocation of space for this array. The user can either declare an array of a large size, say 1000, which is of the type `SDDAS_FLOAT` (e.g., `SDDAS_FLOAT hold_values[1000]`), or the user can use a memory allocation routine to allocate the precise number of bytes needed to hold this information. The number of bytes needed is dependent upon the type of data being requested. The following table illustrates the number of bytes needed for the various data types. The names appearing in bold text represent elements from the `idf_data` structure.

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>NUMBER OF BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR</td>
<td><code>sizeof (SDDAS_FLOAT) * num_sample</code></td>
</tr>
<tr>
<td>SWEEP_STEP</td>
<td><code>sizeof (SDDAS_FLOAT) * num_swp_steps</code></td>
</tr>
<tr>
<td>CAL_DATA</td>
<td><code>sizeof (SDDAS_FLOAT) * num_sample</code></td>
</tr>
<tr>
<td>MODE</td>
<td><code>sizeof (SDDAS_FLOAT) * 1</code></td>
</tr>
<tr>
<td>D_QUAL</td>
<td><code>sizeof (SDDAS_FLOAT) * 1</code></td>
</tr>
<tr>
<td>PITCH_ANGLE</td>
<td><code>sizeof (SDDAS_FLOAT) * num_pitch</code></td>
</tr>
<tr>
<td></td>
<td>or <code>sizeof (SDDAS_FLOAT) * num_sample</code></td>
</tr>
<tr>
<td>START_AZ_ANGLE</td>
<td><code>sizeof (SDDAS_FLOAT) * num_angle</code></td>
</tr>
<tr>
<td>STOP_AZ_ANGLE</td>
<td><code>sizeof (SDDAS_FLOAT) * num_angle</code></td>
</tr>
<tr>
<td>SC_POTENTIAL</td>
<td><code>sizeof (SDDAS_FLOAT) * num_potential</code></td>
</tr>
<tr>
<td>BACKGROUND</td>
<td><code>sizeof (SDDAS_FLOAT) * num_background</code></td>
</tr>
</tbody>
</table>

If pitch angle data is being requested, a check must be made to see if pitch angle data was calculated. If `num_pitch` is equal to zero, no pitch angles were computed. In this case, the user must allocate `sizeof (SDDAS_FLOAT) * num_sample` bytes since this module will return `num_sample` values, all set to `-3.4e38` (OUTSIDE_MIN). The same check must be made for spacecraft potential and background data. If `num_potential` is equal to zero, no spacecraft potential values were returned. In this case, the user must allocate `sizeof`
(SDDAS_FLOAT) * num_sample bytes since this module will return num_sample values, all set to -3.4e38 (OUTSIDE_MIN). If num_background is equal to zero, no background values were returned. In this case, the user must allocate sizeof (SDDAS_FLOAT) * num_sample bytes since this module will return num_sample values, all set to -3.4e38 (OUTSIDE_MIN). If the user is dynamically allocating the space for the data, it is imperative that the user check these length indicators to ensure that enough space is allocated to hold the data. For example, if the previous call to the read_drec routine set num_sample to 12, the user would have allocated 12 * sizeof (SDDAS_FLOAT) bytes. However, if the next call to the read_drec routine sets num_sample to 18, the user needs to reallocate the memory to 18 * sizeof (SDDAS_FLOAT) bytes. If the user does not reallocate the memory for the data, the convert_to_units routine will attempt to write into memory beyond the data array, which could result in abnormal program termination. It is advised that the user reallocate space only if the number of data elements increases since the memory allocation subroutines can become time and resource consuming if called after every read_drec call.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
read_drec 1R
get_data_key 1R
get_version_number 1R
create_data_structure 1R
create_idf_data_structure 1R
ret_codes 1H
user_defs 1H
libbase_idfs 1H
idf_data 1S

BUGS
None

EXAMPLES
Convert one sweep of data from sensor 2 in the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. Return the sensor data in raw units and do not check for fill data values. Assumption is that no more than 1000 values are returned.

#include "libbase_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_Ulong data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT hold_values[1000];
SDDAS_LONG *tbl_oper;
SDDAS_SHORT status;
SDDAS_CHAR num_tbs, *tbls_to_apply;
void *idf_data_ptr;

num_tbs = 0;
tbls_to_apply = NULL;
tbl_oper = NULL;
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);

if (status != ALL_OKAY)
{
    printf ("n Error %d returned by get_data_key routine.n", status);
    exit (-1);
}

get_version_number (vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by create_idf_data_structure routine.n", status);
    exit (-1);
}

status = read_drec (data_key, ",", vnum, idf_data_ptr, 2, 1, 1);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by read_drec routine.n", status);
    exit (-1);
}

status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, SENSOR, 0, num_tbs,
tbls_to_apply, tbl_oper, hold_values, 0, 0);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by convert_to_units routine.n", status);
    exit (-1);
}

Convert sensor data, sweep step data, calibration data, data quality data, pitch angle data,
azimuthal angle data, spacecraft potential data and background data for sensor 2 in addition
to mode data for instrument status value 0 in the virtual instrument RTLA, which is part of
the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified
with the TSS project. Allocate the data arrays so that no memory waste is encountered.

#include "libbase_idfs.h"
#include "user_defs.h"
#include "ret_codes.h"

struct idf_data *EXP_DATA;
SDDAS_UINT data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *data_values, *swp_values, *cal_values, *mode_values, dqual_value;
SDDAS_FLOAT *pitch_values, *start_az_values, *stop_az_value, *potential_values;
SDDAS_FLOAT *background;
SDDAS_LONG offset, *tbl_oper;
SDDAS_SHORT status;
SDDAS_CHAR cset, *tbls_to_apply, num_tbls;
size_t bytes;
void *base_potential, *base_background, *idf_data_ptr;

num_tbls = 0;
tbls_to_apply = NULL;
tbl_oper = NULL;
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}
EXP_DATA = (struct idf_data *) idf_data_ptr;

status = read_drec (data_key, ",", vnum, idf_data_ptr, 2, 1, 1);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by read_drec routine.\n", status);
    exit (-1);
}
bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_sample;
if ((base_sen = malloc (bytes)) == NULL)
{
    printf ("\n No memory for array that holds converted sensor data.\n");
    return (-1);
}

/*  Return data values in raw units and check for fill values of 255 (raw telemetry value). */
data_values = (SDDAS_FLOAT *) base_sen;
status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, SENSOR, 0, num_tbls,
tbls_to_apply, tbl_oper, data_values, 1, 255);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by convert_to_units routine.\n", status);
    exit (-1);
}

bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_swp_steps;
if ((base_swp = malloc (bytes)) == NULL)
{
    printf ("\n No memory for array that holds converted sweep data.\n");
    return (-1);
}

/*  Return raw step values. */
swp_values = (SDDAS_FLOAT *) base_swp;
status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, SWEEP_STEP, 0,
    num_tbls, tbls_to_apply, tbl_oper, swp_values, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by convert_to_units routine.\n", status);
    exit (-1);
}

bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_sample;
if ((base_cal = malloc (bytes)) == NULL)
{
    printf ("\n No memory for array that holds converted cal. data.\n");
    return (-1);
}
/* Return raw calibration values. */

cal_values = (SDDAS_FLOAT *) base_cal;
cset = 1;
status = convert_to_units (data_key, "", vnum, idf_data_ptr, 2, CAL_DATA, cset,
num_tbls, tbls_to_apply, tbl_oper, cal_values, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by convert_to_units routine.\n", status);
    exit (-1);
}

bytes = sizeof (SDDAS_FLOAT) * 1;
if ((base_mode = malloc (bytes)) == NULL)
{
    printf ("\n No memory for array that holds converted mode data.\n");
    return (-1);
}

/* Return raw mode values. Notice that for the sensor parameter, */
/* we are passing in a zero, which is the mode of interest. */

mode_values = (SDDAS_FLOAT *) base_mode;
status = convert_to_units (data_key, "", vnum, idf_data_ptr, 0, MODE, 0, num_tbls,
tbls_to_apply, tbl_oper, mode_values, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by convert_to_units routine.\n", status);
    exit (-1);
}

/* Return data quality values. */

status = convert_to_units (data_key, "", vnum, idf_data_ptr, 2, D_QUAL, 0,
num_tbls, tbls_to_apply, tbl_oper, &dqual_value, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by convert_to_units routine.\n", status);
    exit (-1);
}

if (EXP_DATA->num_pitch == 0)
    bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_sample;
else
    bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_pitch;
if (((base_pitch = malloc (bytes)) == NULL)
    {
        printf ("\n No memory for array that holds pitch angle data.");
        return (-1);
    }

/* Return pitch angle data values in raw units (by default, degrees). */
pitch_values = (SDDAS_FLOAT *) base_pitch;
status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, PITCH_ANGLE, 0,
    num_tbs, tbs_to_apply, tbl_oper, pitch_values, 0, 0);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by convert_to_units routine.\n", status);
        exit (-1);
    }

bytes = 2 * sizeof (SDDAS_FLOAT) * EXP_DATA->num_angle;
if (((base_az = malloc (bytes)) == NULL)
    {
        printf ("\n No memory for array that holds azimuthal angle data.");
        return (-1);
    }

/* Return azimuthal angle data values in raw units (by default, degrees). */
/* Cast base_az in setting of stop_az_values since it is a void pointer. */
start_az_values = (SDDAS_FLOAT *) base_az;
offset = sizeof (SDDAS_FLOAT) * EXP_DATA->num_angle;
stop_az_values = (SDDAS_FLOAT *) ((SDDAS_CHAR *) base_az + offset);

status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, START_AZ_ANGLE,
    0, num_tbs, tbs_to_apply, tbl_oper, start_az_values, 0, 0);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by convert_to_units routine.\n", status);
        exit (-1);
    }

status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, STOP_AZ_ANGLE, 0,
    num_tbs, tbs_to_apply, tbl_oper, stop_az_values, 0, 0);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by convert_to_units routine.\n", status);
        exit (-1);
    }
if (EXP_DATA->num_potential == 0)
    bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_sample;
else
    bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_potential;
if ((base_potential = malloc (bytes)) == NULL)
{
    printf ("\nNo memory for array that holds spacecraft potential data."");
    return (-1);
}

/* Return spacecraft potential data values in raw units. */

potential_values = (SDDAS_FLOAT *) base_potential;
status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, SC_POTENTIAL, 0,
                          num_tbls, tbls_to_apply, tbl_oper, potential_values, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\nError %d returned by convert_to_units routine.\n", status);
    exit (-1);
}

if (EXP_DATA->num_background == 0)
    bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_sample;
else
    bytes = sizeof (SDDAS_FLOAT) * EXP_DATA->num_background;
if ((base_background = malloc (bytes)) == NULL)
{
    printf ("\nNo memory for array that holds background data."");
    return (-1);
}

/* Return background data values in raw units. */

background_values = (SDDAS_FLOAT *) base_background;
status = convert_to_units (data_key, ",", vnum, idf_data_ptr, 2, BACKGROUND, 0,
                          num_tbls, tbls_to_apply, tbl_oper, background_values, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\nError %d returned by convert_to_units routine.\n", status);
    exit (-1);
}
CREATE_DATA_STRUCTURE
function - creates an instance of the structure that is needed to hold the data for the data set specified

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT create_data_structure (SDDAS_ULONG data_key,  
      SDDAS_CHAR *exten,  
      SDDAS_USHORT version, void **data_ptr)

ARGUMENTS
data_key  - unique value which indicates the data set of interest
exten   - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version   - IDFS data set identification number which allows for multiple
openings of the same data set
data_ptr   - pointer to the newly created data structure
create_data_structure - routine status (see TABLE 1)

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_DSTR_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the data structure was allocated</td>
</tr>
</tbody>
</table>

error codes returned by create_idf_data_structure ()
error codes returned by create_tensor_data_structure ()

DESCRIPTION
Create_data_structure is the IDFS routine that creates the appropriate instance of the data structure that is used by the IDFS software to return sensor data and pertinent ancillary data for the data set of interest. The type of data structure allocated depends upon the IDFS data source being processed – either conventional IDFS data with a dependency based upon a scanning variable (refer to create_idf_data_structure) or multi-dimensional tensor IDFS data (refer to create_tensor_data_structure). With each call to this module, a new data structure is created and the address of this structure is returned. In order to access the elements within the data structure, the user must explicitly cast the returned void pointer to a pointer of the correct type.

In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_data_structure routine N times to create N instances of the data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure. The contents of this structure is described in section 1S of the IDFS Programmers Manual.
The address of each data structure that is created is kept and this memory is freed when the free_experiment_info routine is called. The user must not free the memory themselves since the IDFS software will attempt to free the memory location and the result is uncertain.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
- free_experiment_info 1R
- create_idf_data_structure 1R
- create_tensor_data_structure 1R
- ret_codes 1H
- libbase_idfs 1H
- idf_data 1S
- tensor_data 1S

BUGS
None

EXAMPLES
Create one instance of the data structure that is needed for the data set specified and return the address in the specified parameter. Assume that the data_key and version number have already been set by the appropriate routines.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
void *data_ptr;

status = create_data_structure (data_key, "", vnum, &data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_data_structure routine.\n", status);
    exit (-1);
}
```
CREATE_IDF_DATA_STRUCTURE
function - creates an instance of the idf_data structure

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT  create_idf_data_structure (void **idf_data_ptr)

ARGUMENTS
idf_data_ptr - pointer to the newly created idf_data structure
create_idf_data_structure - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for CREATE_IDF_DATA_STRUCTURE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_DATA_ALL_MALLOC</td>
<td>no memory to hold the address of all allocated idf_data structures</td>
</tr>
<tr>
<td>CREATE_DATA_ALL_REALLOC</td>
<td>no memory for expansion of the area that holds the address of all allocated idf_data structures</td>
</tr>
<tr>
<td>CREATE_DATA_MALLOC</td>
<td>no memory for the idf_data structure</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the idf_data structure was allocated</td>
</tr>
</tbody>
</table>

DESCRIPTION
Create_idf_data_structure is the IDFS routine that creates an instance of the idf_data structure that is used by the IDFS software to return sensor data and pertinent ancillary data for the data set of interest. With each call to this module, a new idf_data structure is created and the address of this structure is returned. In order to access the elements within the idf_data structure, the user must explicitly cast the returned void pointer to a pointer of the type struct idf_data.

In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_idf_data_structure routine N times to create N instances of the idf_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure. The contents of this structure is described in section 1S of the IDFS Programmers Manual.

The address of each idf_data structure that is created is kept and this memory is freed when the free_experiment_info routine is called. The user **must not** free the memory themselves since the IDFS software will attempt to free the memory location and the result is uncertain.
ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
free_experiment_info 1R
ret_codes 1H
libbase_idfs 1H
idf_data 1S

BUGS
None

EXAMPLES
Create one instance of the idf_data structure and return the address in the specified parameter. Cast the returned void pointer so that elements of the idf_data structure can be referenced.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

struct idf_data *EXP_DATA;
SDDAS_SHORT status;
void *idf_data_ptr;

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}
EXP_DATA = (struct idf_data *) idf_data_ptr;

/* Print the start time. */

printf("\n START TIME_MS = %d", EXP_DATA->bmilli);
```
CREATE_TENSOR_DATA_STRUCTURE
function - creates an instance of the tensor_data structure

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT create_tensor_data_structure (void **tensor_data_ptr)

ARGUMENTS
 tensor_data_ptr - pointer to the newly created tensor_data structure
 create_tensor_data_structure - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for CREATE_TENSOR_DATA_STRUCTURE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_TENSOR_DATA_ALL_MALLOC</td>
<td>no memory to hold the address of all allocated tensor_data structures</td>
</tr>
<tr>
<td>CREATE_TENSOR_DATA_ALL_REALLOC</td>
<td>no memory for expansion of the area that holds the address of all allocated tensor_data structures</td>
</tr>
<tr>
<td>CREATE_TENSOR_DATA_MALLOC</td>
<td>no memory for the tensor_data structure</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the tensor_data structure was allocated</td>
</tr>
</tbody>
</table>

DESCRIPTION
Create_tensor_data_structure is the IDFS routine that creates an instance of the tensor_data structure that is used by the IDFS software to return sensor data and pertinent ancillary data for the multi-dimensional IDFS data set of interest. With each call to this module, a new tensor_data structure is created and the address of this structure is returned. In order to access the elements within the tensor_data structure, the user must explicitly cast the returned void pointer to a pointer of the type struct tensor_data.

In most cases, one data structure is sufficient to process any number of distinct multi-dimensional IDFS data sets. However, if more than one structure is needed, the user may call the create_tensor_data_structure routine N times to create N instances of the tensor_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure. The contents of this structure is described in section 1S of the IDFS Programmers Manual.

The address of each tensor_data structure that is created is kept and this memory is freed when the free_experiment_info routine is called. The user must not free the memory themselves since the IDFS software will attempt to free the memory location and the result is uncertain.
ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
free_experiment_info 1R
ret_codes 1H
libbase_idfs 1H
tensor_data 1S

BUGS
None

EXAMPLES
Create one instance of the tensor_data structure and return the address in the specified parameter. Cast the returned void pointer so that elements of the tensor_data structure can be referenced.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

struct tensor_data *TENSOR_DATA;
SDDAS_SHORT status;
void *tensor_data_ptr;

status = create_tensor_data_structure (&tensor_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_tensor_data_structure routine.\n", status);
    exit (-1);
}
TENSOR_DATA = (struct tensor_data *) tensor_data_ptr;

/* Print the start time. */
printf ("\n START TIME_MS = %d", TENSOR_DATA->bmilli);
```
**DESTROY_LAST_IDF_DATA_STRUCTURE**

function – free the last instance of the *idf_data* structure allocated

**SYNOPSIS**

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT destroy_last_idf_data_structure ()
```

**ARGUMENTS**

- `destroy_last_idf_data_structure` - routine status (see TABLE 1)

**TABLE 1. Status Codes Returned for DESTROY_LAST_IDF_DATA_STRUCTURE**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESTROY_NO_IDF_DATA</td>
<td>there are no <em>idf_data</em> structures to be freed</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the last allocated <em>idf_data</em> structure was freed</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

`destroy_last_idf_data_structure` is the IDFS routine that destroys or frees the last instance of the *idf_data* structure that was allocated by the `create_idf_data_structure` routine in order to return sensor data and pertinent ancillary data for the data set of interest. An array of pointers that holds the address of each *idf_data* structure that is created is kept and with each call to this module, the last *idf_data* structure that was allocated is freed. If this module is called more times than the number of allocated structures, an error code is returned to alert the calling routine of this situation.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `create_idf_data_structure` 1R
- `ret_codes` 1H
- `libbase_idfs` 1H
- `idf_data` 1S

**BUGS**

None
EXAMPLES

Destroy, or free, the last instance of the **idf_data** structure.

```
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT status;

status = destroy_last_idf_data_structure ();
if (status != ALL_OKAY)
{
    printf ("Error %d returned by destroy_last_idf_data_structure routine.\n", status);
    exit (-1);
}
```
DESTROY_LAST_TENSOR_DATA_STRUCTURE
function – free the last instance of the tensor_data structure allocated

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT destroy_last_tensor_data_structure ()

ARGUMENTS
destroy_last_tensor_data_structure - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for DESTROY_LAST_TENSOR_DATA_STRUCTURE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESTROY_NO_TENSOR_DATA</td>
<td>there are no tensor_data structures to be freed</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the last allocated tensor_data structure was freed</td>
</tr>
</tbody>
</table>

DESCRIPTION
Destroy_last_tensor_data_structure is the IDFS routine that destroys or frees the last instance of the tensor_data structure that was allocated by the create_tensor_data_structure routine in order to return sensor data and pertinent ancillary data for the multi-dimensional IDFS data set of interest. An array of pointers that holds the address of each tensor_data structure that is created is kept and with each call to this module, the last tensor_data structure that was allocated is freed. If this module is called more times than the number of allocated structures, an error code is returned to alert the calling routine of this situation.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
create_tensor_data_structure 1R
ret_codes 1H
libbase_idfs 1H
tensor_data 1S

BUGS
None
EXAMPLES

Destroy, or free, the last instance of the tensor_data structure.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT status;

status = destroy_last_tensor_data_structure ();
if (status != ALL_OKAY)
{
    printf ("%d Error %d returned by destroy_last_tensor_data_structure routine.\n", status);
    exit (-1);
}
```
extract_single_element_from_idfs_tensor (1R)

**EXTRACT_SINGLE_ELEMENT_FROM_IDFS_TENSOR**

function – extracts a single element from the specified IDFS tensor

**SYNOPSIS**

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

void extract_single_element_from_idfs_tensor (SDDAS_SHORT tensor_rank,
                                           SDDAS_ULONG *tensor_next_dimen, void *tensorA,
                                           SDDAS_ULONG *element_indices, void *ret_ptr,
                                           SDDAS_CHAR long_dtype)
```

**ARGUMENTS**

- `tensor_rank` - the rank of the tensor being processed
- `tensor_next_dimen` - pointer to an array of size `tensor_rank` that holds the number of data values to bypass in order to get to the next index for a given dimension ([0] = first dimension or slowest varying dimension)
- `tensorA` - pointer to the input tensor data
- `element_indices` - pointer to an array of size `tensor_rank` that holds the indices for each of the dimensions defined so that a single element of the tensor can be indexed
- `ret_ptr` - pointer to the resultant
- `long_dtype` - flag indicating whether the input / return values are integer or floating point
  - 0 - values are floating point values
  - 1 - values are integer values

**DESCRIPTION**

`extract_single_element_from_idfs_tensor` is the IDFS routine that is used to extract a single element from the multi-dimensional IDFS data that is returned by the `read_tensor_data` module. The first two arguments, `tensor_rank` and `tensor_next_dimen` can be taken directly from the `tensor_data` structure that is returned by the `read_tensor_data` module. The argument `element_indices` is used to specify the start index location at which the extraction is to take place for each defined dimension for the multi-dimensional data held by the `tensorA` argument.

For the time being, multi-dimensional IDFS data can not be dynamically converted to any other physical unit; therefore, the data must be stored in the physical unit desired when the data set is created. However, the `read_tensor_data` module will return two sets of data within the `tensor_data` structure. One set represents the raw integer values that are stored within the data record and one set represents the floating point values that result when transferring the raw integer values into the data type defined by `d_type` in the VIDF file for the IDFS data set being processed. The two arguments, `tensorA` and `long_dtype`, should be coupled so that the user can extract data from either of these two data sets correctly.
extract_single_element_from_idfs_tensor (1R)

ERRORS
This routine returns no status or error codes.

SEE ALSO
create_data_structure 1R
create_tensor_data_structure 1R
read_tensor_data 1R
libbase_idfs 1H
tensor_data 1S

BUGS
None

EXAMPLES
Extract the data, one element at a time, from the multi-dimensional data returned by the read_tensor_data module. It is already known that the data stored in the multi-dimensional IDFS data set is a 2-D tensor.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

struct tensor_data *TENSOR_DATA;
register SDDAS_USHORT i, j;
SDDAS_FLOAT conv_data;
SDDAS ULONG data_key;
SDDAS USHORT version;
SDDAS ULONG element_indices[IDFS_MAX_DIMEN];
SDDAS ULONG data_val;
SDDAS SHORT status, sensor = 0;
SDDAS CHAR extension[3];
void *tensor_data_ptr, *param;

status = get_data_key ("MARS", "Mars_Express", "ASPERA-3", NPD", "NPD1BM16",
&data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d from get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&version);
strcpy (extension,"");
```
status = create_tensor_data_structure (&tensor_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_tensor_data_structure routine.\n", status);
    exit (-1);
}

TENSOR_DATA = (struct tensor_data *) tensor_data_ptr;
status = read_tensor_data (data_key, extension, version, tensor_data_ptr, sensor, 1);
if (status < 0)
{
    printf ("\n Error %d from read_tensor_data.\n", status);
    exit (-1);
}

if (status == ALL_OKAY || TENSOR_DATA->filled_data)
{
    /*********************************************************************/
    /*  Print the raw integer data values, 6 values per row, in exponential format.              */
    /*********************************************************************/
    for (i = 0; i < TENSOR_DATA->tensor_sizes[0]; ++i)
    {
        for (j = 0; j < TENSOR_DATA->tensor_sizes[1]; ++j)
        {
            element_indices[0] = i;
            element_indices[1] = j;
            param = &data_val;

            extract_single_element_from_idfs_tensor (TENSOR_DATA->tensor_rank,
                                                    TENSOR_DATA->tnext_dimen,
                                                    (void *) TENSOR_DATA->sen_data,
                                                    &element_indices[0], param, 1);

            if (j == 0)
                printf ("\n");
            printf ("%ld ", data_val);
        }
    }
for (i = 0; i < TENSOR_DATA->tensor_sizes[0]; ++i)
{
    for (j = 0; j < TENSOR_DATA->tensor_sizes[1]; ++j)
    {
        element_indices[0] = i;
        element_indices[1] = j;
        param = &conv_data;
        extract_single_element_from_idfs_tensor (TENSOR_DATA->tensor_rank,
                                                TENSOR_DATA->tnext_dimen,
                                                (void *) TENSOR_DATA->tdata,
                                                &element_indices[0], param, 0);

        if (j == 0)
            printf ("\n");
        printf ("%.4f ", conv_data);
    }
}

FIELDS_TO_KEY
    function - create a key value which identifies the data set of interest

SYNOPSIS
    #include "libbase_idfs.h"
    #include "libCfg.h"

    void fields_to_key (SDDAS_SHORT *params, SDDAS_ULONG *data_key)

ARGUMENTS
  params  - an array which holds the assigned database numbers for the project, mission, experiment, instrument and virtual instrument to be accessed
  data_key - unique value which indicates the data set being requested

DESCRIPTION
    Fields_to_key is the IDFS routine which creates a key that reflects the data set being accessed by utilizing the assigned database numbers for the project, mission, experiment, instrument and virtual instrument of interest. The IDFS routine get_data_key performs the same function but works with the assigned database names instead of the assigned database numbers. Most of the IDFS routines utilize key values; therefore, a call to either this routine or to the get_data_key routine must be made before any of the other IDFS routines that utilize a key value can be called.

    The user selects the data set of interest by specifying a virtual instrument from a specific instrument, which comes from a parent experiment within a mission which is associated with a specific project. All references for these items are through assigned database numbers. The params parameter is an array that holds these assigned database numbers in the order specified below:

    element 0    project identification number
    element 1    mission identification number
    element 2    experiment identification number
    element 3    instrument identification number
    element 4    virtual instrument identification number

ERRORS
    This routine returns no status or error codes.

SEE ALSO
    get_data_key  1R

BUGS
    None
EXAMPLES
Retrieve the data key for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libbase_idfs.h"
#include "libCfg.h"

extern LinkList Projects;

StrHier Scode;
SDDAS_ULONGLONG data_key;
SDDAS_SHORT project, mission, exper, inst, vinst, params[5];

Scode = SourceByStr (Projects, "TSS", (SDDAS_CHAR *) 0);
if (Scode == NULL)
{
    printf ("\n Error in calling SourceByStr for project number.\n");
    exit (-1);
}
else
    project = SNUM (Scode);

Scode = SourceByStr (Projects, "TSS", "TSS-1", (SDDAS_CHAR *) 0);
if (Scode == NULL)
{
    printf ("\n Error in calling SourceByStr for mission number.\n");
    exit (-1);
}
else
    mission = SNUM (Scode);

Scode = SourceByStr (Projects, "TSS", "TSS-1", "RETE", (SDDAS_CHAR *) 0);
if (Scode == NULL)
{
    printf ("\n Error in calling SourceByStr for experiment number.\n");
    exit (-1);
}
else
    exper = SNUM (Scode);

Scode = SourceByStr (Projects, "TSS", "TSS-1", "RETE", "RETE", (SDDAS_CHAR *) 0);
if (Scode == NULL)
{
    printf ("\n Error in calling SourceByStr for instrument number.\n");
```
In the coding example provided, the database assignment numbers for the project, mission, experiment, instrument and virtual instrument are retrieved through calls to the modules `SourceByStr` and `SNUM`. A detailed description of these modules can be found in the IDFS Configuration Definition Document.
FILE_OPEN
function - open IDFS files (data, header and VIDF) for the time period requested

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT file_open (SDDAS_ULONG data_key, SDDAS_CHAR *exten, 
SDDAS_USHORT version, SDDAS_SHORT btime_yr, 
SDDAS_SHORT btime_day, SDDAS_LONG btime_sec, 
SDDAS_LONG btime_nano, SDDAS_SHORT etime_yr, 
SDDAS_SHORT etime_day, SDDAS_LONG etime_sec, 
SDDAS_LONG etime_nano, SDDAS_CHAR mode_data)

ARGUMENTS
data_key - unique value which indicates the data set of interest
exten  - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple
            openings of the same data set
btime_yr - beginning year for data being requested
btime_day - beginning day of year for data being requested
btime_sec - beginning time of day in seconds for data being requested
btime_nano - beginning time of day residual in nanoseconds
etime_yr - ending year for data being requested
etime_day - ending day of year for data being requested
etime_sec - ending time of day in seconds for data being requested
etime_nano - ending time of day residual in nanoseconds
mode_data - flag indicating if instrument status (mode) data will be requested
            for this data set
            0    - instrument status data will not be requested for this data set
            1    - instrument status data will be requested for this data set
file_open - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for FILE_OPEN

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_DATA</td>
<td>there is no data available for the requested time period (playback scenario)</td>
</tr>
<tr>
<td>OPEN_PTR_MALLOC</td>
<td>no memory for IDFS location pointers</td>
</tr>
<tr>
<td>OPEN_EX_REALLOC</td>
<td>no memory for experiment definition structure expansion</td>
</tr>
<tr>
<td>ALL_FLAG_MALLOC</td>
<td>no memory for sensor flags</td>
</tr>
<tr>
<td>RTIME_NO HEADER</td>
<td>header file could not be opened (real-time scenario)</td>
</tr>
<tr>
<td>RTIME_NO DATA</td>
<td>data file could not be opened (real-time scenario)</td>
</tr>
<tr>
<td>PBACK_NO HEADER</td>
<td>header file could not be opened (playback scenario)</td>
</tr>
<tr>
<td>PBACK_NO DATA</td>
<td>data file could not be opened (playback scenario)</td>
</tr>
<tr>
<td>ONCE_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ONCE_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>ONCE_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>ONCE_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>ONCE_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>ONCE_BAD_HEADER_FMT</td>
<td>invalid header record format value</td>
</tr>
<tr>
<td>ONCE_BAD_TENSOR_RANK</td>
<td>invalid rank value for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_TENSOR_LENGTHS</td>
<td>invalid dimension length value for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_CTARGET_MALLOC</td>
<td>no memory for cal_target VIDF values</td>
</tr>
<tr>
<td>ONCE_CLEN_MALLOC</td>
<td>no memory for cal_wlen VIDF values</td>
</tr>
<tr>
<td>ONCE_CSCOPE_MALLOC</td>
<td>no memory for cal_scope VIDF values</td>
</tr>
<tr>
<td>CAL_DATA_MALLOC</td>
<td>no memory for calibration information</td>
</tr>
<tr>
<td>ONCE_DATA_MALLOC</td>
<td>no memory for data record information</td>
</tr>
<tr>
<td>ONCE_TBL_INFO_MALLOC</td>
<td>no memory for structures which hold non-array table specific information</td>
</tr>
<tr>
<td>ONCE_D_TYPE_MALLOC</td>
<td>no memory for d_type VIDF values</td>
</tr>
<tr>
<td>ONCE_TDW_LEN_MALLOC</td>
<td>no memory for tdw_len VIDF values</td>
</tr>
<tr>
<td>ONCE_SPIN_OFF_MALLOC</td>
<td>no memory for spin_time_off VIDF values</td>
</tr>
<tr>
<td>ONCE_SEN_STAT_MALLOC</td>
<td>no memory for sen_status VIDF values</td>
</tr>
<tr>
<td>ONCE_CDTYPE_MALLOC</td>
<td>no memory for cal_d_type VIDF values</td>
</tr>
<tr>
<td>ONCE_BAD_NUM_TBLS</td>
<td>num_tls can not be set to any value other than 0 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_CAL_TARGET</td>
<td>cal_target can not be set to any value other than 0 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_MAX_NSS</td>
<td>max_nss can not be set to any value other than 1 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_SMP_ID</td>
<td>smp_id must be set to 3 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_DA_METHOD</td>
<td>da_method must be set to 0 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_SWP_LEN</td>
<td>swp_len must be set to 1 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ONCE_BAD_SEN_MODE</td>
<td>sen_mode must be set to either 3 or 7 for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>NUM_CAL_REALLOC</td>
<td>no memory for expansion of calibration array</td>
</tr>
<tr>
<td>UPDATE_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>UPDATE_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>UPDATE_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>UPDATE_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>UPDATE_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>UPDATE_IDF_NO_FILL</td>
<td>a fill value must be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_PA_DEF</td>
<td>pitch angle can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_POT_DEF</td>
<td>spacecraft potential data can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_SPIN_DEF</td>
<td>start of spin data source can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_PMI_DEF</td>
<td>euler angle can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_CP_DEF</td>
<td>celestial position angles can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_BKGD_DEF</td>
<td>background data can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>ASCII_AFTER_SENSOR</td>
<td>all ASCII and mode-dependent tables must be defined after all other tables are defined in the VIDF</td>
</tr>
<tr>
<td>CONST_ANG_MALLOC</td>
<td>no memory for angle offset values</td>
</tr>
<tr>
<td>CONST_TEMP_MALLOC</td>
<td>no memory for temporary working area</td>
</tr>
<tr>
<td>CONST_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
</tbody>
</table>
### DESCRIPTION

**File_open** is the IDFS file open routine. The data set of interest is referenced through the key value **data_key** which can be created using the **get_data_key** module. The files that are opened, the header, data, and VIDF file, are dependent on the data set (**data_key**), file name extension (**exten**) and the time range specified. The maximum number of files that can be opened at one time is a system dependent value. For example, with SunOS, the maximum number of open file descriptors is set at 256. This value can be modified; however, system performance may be degraded as this value is increased.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. In either case, the specified IDFS data set will only be opened once for each unique parameter set. If additional calls are made to this routine with the same parameter set, the module simply returns the **ALL_OKAY** status code. The user should call the **get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The appropriate files are searched for within the current on-line database for the data set designated. The database returns both the file name and a status code. The database contains information about the satellite data on the local machine. The contents of the database entries include the file name referencing the header file, data file or the VIDF file. For playback data, the contents of the database entries also include the name of the primary and/or secondary media names along with their respective data sizes and offsets on the media. These entries are indexed by increasing universal time to expedite data searching. If the files do exist on the local machine, the files are opened and the file descriptors are saved in an internally defined structure for later use by the other IDFS routines. If the files do not exist on the local machine, the appropriate error code is returned.
This routine opens the first set of files within the time span over which data is to be processed. For a real-time scenario, there is only one set of files for the time span being processed. In post acquisition analysis, if there is more than one file set within the requested time interval, the remaining files will be opened and processed after the currently opened files are processed.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

Once the files have been opened successfully, the VIDF is loaded into memory. Selected information from the VIDF file is retrieved and stored in an internally defined structure. The fields selected are those pieces of information that are to be used by the other IDFS routines. After the necessary elements have been read from the VIDF file, internal flags are set to indicate that all sensors associated with the data set are to be processed and memory to hold various information concerning each sensor is to be allocated by the routine `file_pos`. If only a few of the sensors associated with the data set are to be processed, these can be selected by the use of the routine `select_sensor`, which will reset the internal flags such that only the necessary sensors will be processed and have space allocated.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_pos` 1R
- `select_sensor` 1R
- `get_data_key` 1R
- `get_version_number` 1R
- `ret_codes` 1H
- `libbase_idfs` 1H

**BUGS**

None
EXAMPLES
Open the real-time files associated with the virtual instrument RTLA, which is part of the
RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with
the TSS project. In the calling sequence, indicate that instrument status data is not to be
requested for the data set in question.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = file_open (data_key, ",", vnum, -1, -1, -1, 0, -1, -1, -1, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_open routine.\n", status);
    exit (-1);
}
```

The RTLA default data set was modified to remove all counts less than 2. This new data set
resides in the user's home directory and has the 2 character extension L2 appended to the
IDFS file names. In the calling sequence, indicate that instrument status data is to be
requested for the data set in question. To open this data set:

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
```
get_version_number (&vnum);

status = file_open (data_key, "L2", vnum, -1, -1, 0, -1, -1, -1, 1);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by file_open routine.\n", status);
    exit (-1);
}
FILE_POS
function - allocates memory for necessary idfs data structures and positions the file pointers at the requested time in the files.

SYNOPSIS

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT file_pos (SDDAS_ULONG data_key, SDDAS_CHAR *exten, SDDAS_USHORT version, void *data_ptr, SDDAS_SHORT btime_yr, SDDAS_SHORT btime_day, SDDAS_LONG btime_sec, SDDAS_LONG btime_nano, SDDAS_SHORT etime_yr, SDDAS_SHORT etime_day, SDDAS_LONG etime_sec, SDDAS_LONG etime_nano)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten  - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version  - IDFS data set identification number which allows for multiple openings of the same data set
data_ptr - pointer to the data structure that is to hold sensor data and pertinent ancillary data for the data set of interest (either idf_data or tensor_data)
btime_yr - beginning year for data being requested
btime_day - beginning day of year for data being requested
btime_sec - beginning time of day in seconds for data being requested
btime_nano - beginning time of day residual in nanoseconds
etime_yr - ending year for data being requested
etime_day - ending day of year for data being requested
etime_sec - ending time of day in seconds for data being requested
etime_nano - ending time of day residual in nanoseconds
file_pos - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for FILE_POS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>POS_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>POS_DATA_READ_ERROR</td>
<td>read error on data file</td>
</tr>
<tr>
<td>SCOM_TBL_MALLOC</td>
<td>no memory for table offset value for sensors being processed</td>
</tr>
<tr>
<td>SCOM_PTR_MALLOC</td>
<td>no memory for pointers to the memory allocated to hold table offset values</td>
</tr>
<tr>
<td>SCOM_INDEX_MALLOC</td>
<td>no memory for sensor index array</td>
</tr>
<tr>
<td>SCOM_SEN_PTR_MALLOC</td>
<td>no memory for array of structures which hold sensor-specific information</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SEN_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>SEN_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>SEN_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>SEN_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>SEN_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>CCOM_MATCH_MALLOC</td>
<td>no memory for temporary array used for determining the number of sensor table</td>
</tr>
<tr>
<td></td>
<td>combinations</td>
</tr>
<tr>
<td>CCOM_VAL_MALLOC</td>
<td>no memory for base offset and comparison offset values</td>
</tr>
<tr>
<td>CRIT_ACT_MALLOC</td>
<td>no memory for critical action information</td>
</tr>
<tr>
<td>TBL_MISC_MALLOC</td>
<td>no memory to hold table information for all integer tables</td>
</tr>
<tr>
<td>TBL_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>TBL_IDF_MANY_BYTES</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>TBL_IDF_TBL_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>TBL_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>TBL_VAR_NOT_RAW</td>
<td>the table can only be a function of raw data since the table format specifies an</td>
</tr>
<tr>
<td></td>
<td>expanded look-up table</td>
</tr>
<tr>
<td>TBL_VAR_NOT_CAL</td>
<td>the table can only be a function of a calibration set since the table type is sweep-</td>
</tr>
<tr>
<td></td>
<td>length dependent table and the table format specifies an expanded look-up table</td>
</tr>
<tr>
<td>TBL_MALLOC</td>
<td>no memory to hold values for all integer tables</td>
</tr>
<tr>
<td>GET_ACTION_MALLOC</td>
<td>no memory for array that holds the actions for critical status bytes</td>
</tr>
<tr>
<td>CRIT_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>CRIT_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>CRIT_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>CRIT_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>CRIT_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>MODE_PTR_MALLOC</td>
<td>no memory for array of structures which hold mode-specific information</td>
</tr>
<tr>
<td>MODE_TBL_MISC_MALLOC</td>
<td>no memory to hold table information for all integer mode-dependent tables</td>
</tr>
<tr>
<td>MODE_TBL_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>MODE_TBL_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>MODE_TBL_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>MODE_TBL_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>MODE_TBL_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>MODE_TBL_VAR_NOT_RAW</td>
<td>the table can only be a function of raw data since the table format specifies an</td>
</tr>
<tr>
<td></td>
<td>expanded loop-up table</td>
</tr>
<tr>
<td>MODE_TBL_VAR_NOT_CAL</td>
<td>the table can only be a function of a calibration set since the table type is sweep-</td>
</tr>
<tr>
<td></td>
<td>length dependent table and the table format specifies an expanded look-up table</td>
</tr>
<tr>
<td>MODE_TBL_MALLOC</td>
<td>no memory to hold values for all integer mode-dependent tables</td>
</tr>
<tr>
<td>MODE_TBL_SZ_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>MODE_TBL_SZ_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>MODE_TBL_SZ_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>MODE_TBL_SZ_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>MODE_TBL_SZ_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ALLOC_HDR_READ_ERROR</td>
<td>read error on header file</td>
</tr>
<tr>
<td>ALLOC_HDR_MALLOC</td>
<td>no memory for header record information</td>
</tr>
<tr>
<td>ALLOC_HDR_REALLOC</td>
<td>no memory for header information expansion (header increased in size)</td>
</tr>
<tr>
<td>SWEEP_TIME_MALLOC</td>
<td>no memory for time of sample values</td>
</tr>
<tr>
<td>TIME_OFF_MALLOC</td>
<td>no memory for time offset values for individual sensors</td>
</tr>
<tr>
<td>EXP_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>EXP_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>EXP_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>EXP_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>EXP_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>TIMING_MALLOC</td>
<td>no memory for structures that hold timing information for pixel location</td>
</tr>
<tr>
<td>PBACK_LOS</td>
<td>an LOS indicator record was encountered when trying to find the record containing</td>
</tr>
<tr>
<td></td>
<td>the requested start time (playback scenario only) - user is advised to change the</td>
</tr>
<tr>
<td></td>
<td>start time to either a previous or later time period</td>
</tr>
<tr>
<td>PBACK_NEXT_FILE</td>
<td>a NEXT_FILE indicator record was encountered when trying to find the record</td>
</tr>
<tr>
<td></td>
<td>containing the requested start time (playback scenario only) - user is advised to</td>
</tr>
<tr>
<td></td>
<td>change the start time to either a previous or later time period</td>
</tr>
<tr>
<td>POS_HDR_READ_ERROR</td>
<td>read error on header file</td>
</tr>
<tr>
<td>POS_HDR_MALLOC</td>
<td>no memory for header record information</td>
</tr>
<tr>
<td>POS_HDR_REALLOC</td>
<td>no memory for header information expansion (header increased in size)</td>
</tr>
<tr>
<td>FILE_POS_DATA_GAP</td>
<td>the time range requested lies within a gap found within the data file</td>
</tr>
<tr>
<td>FILE_POS_MODE</td>
<td>error encountered when positioning file descriptors for instrument status</td>
</tr>
<tr>
<td></td>
<td>(mode) data</td>
</tr>
<tr>
<td>FILE_POS_PA</td>
<td>error encountered while trying to position the pitch angle IDFS data set</td>
</tr>
<tr>
<td>FILE_POS_SPIN</td>
<td>error encountered while trying to position the start of spin data source</td>
</tr>
<tr>
<td>FILE_POS_POT</td>
<td>error encountered while trying to position the spacecraft potential IDFS data set</td>
</tr>
<tr>
<td>FILE_POS_EULER</td>
<td>error encountered while trying to position the euler angle IDFS data set</td>
</tr>
<tr>
<td>FILE_POS_CP</td>
<td>error encountered while trying to position the celestial position angle</td>
</tr>
<tr>
<td></td>
<td>IDFS data set</td>
</tr>
<tr>
<td>FILE_POS_BKGD</td>
<td>error encountered while trying to position the background IDFS data set</td>
</tr>
<tr>
<td>RHDR_READ_ERROR</td>
<td>read error on header file</td>
</tr>
<tr>
<td>RHDR_HDR_MALLOC</td>
<td>no memory for header information</td>
</tr>
<tr>
<td>RHDR_HDR_REALLOC</td>
<td>no memory for header information expansion</td>
</tr>
<tr>
<td>PITCH_MALLOC</td>
<td>no memory for structure that holds pitch angle information</td>
</tr>
<tr>
<td>PINFO_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>PINFO_IDF_MANY_BYTES</td>
<td>the number of element being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>PINFO_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>PINFO_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>PINFO_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>PA_UNIT_MALLOC</td>
<td>no memory for normal vector definition for pitch angle data</td>
</tr>
<tr>
<td>PA_DATA_MALLOC</td>
<td>no memory for data and normalization factors for pitch angle data</td>
</tr>
<tr>
<td>PA_TBL_MALLOC</td>
<td>no memory for table number / table operation information for pitch angle data</td>
</tr>
<tr>
<td>PA_UNIT_NORMAL</td>
<td>definition of the normal vector for the pitch angle data is incomplete</td>
</tr>
<tr>
<td>NO_PA_CONSTANT</td>
<td>the pitch angle constants are not defined in the VIDF file</td>
</tr>
<tr>
<td>PA_BAD_SRC</td>
<td>the IDFS data source for the pitch angle data is not a scalar instrument</td>
</tr>
<tr>
<td>BAD_PA_FORMAT</td>
<td>the format specification field for the pitch angle data is invalid</td>
</tr>
<tr>
<td>SPIN_SRC_MALLOC</td>
<td>no memory for structure that holds start of spin data source info</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SPIN_SRC_BAD_SRC</td>
<td>the start of spin data source is non-scalar</td>
</tr>
<tr>
<td>SPIN_SINFO_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>SPIN_SINFO_IDF_MANY_BYTES</td>
<td>the number of element being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>SPIN_SINFO_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>SPIN_SINFO_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>SPIN_SINFO_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>POT_INFO_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>POT_INFO_IDF_MANY_BYTES</td>
<td>the number of element being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>POT_INFO_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>POT_INFO_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>POT_INFO_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>POT_TBL_MALLOC</td>
<td>no memory for table number / table operation information for spacecraft potential data</td>
</tr>
<tr>
<td>POT_BAD_SRC</td>
<td>the IDFS data source for the spacecraft potential data is not a scalar instrument</td>
</tr>
<tr>
<td>BAD_SCPOT_FORMAT</td>
<td>the format specification field for the spacecraft potential data is invalid</td>
</tr>
<tr>
<td>POT_MALLOC</td>
<td>no memory for structure that holds spacecraft potential information</td>
</tr>
<tr>
<td>POT_DATA_MALLOC</td>
<td>no memory for data and normalization factors for spacecraft potential data</td>
</tr>
<tr>
<td>EULER_INFO_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>EULER_INFO_IDF_MANY_BYTES</td>
<td>the number of element being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>EULER_INFO_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>EULER_INFO_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>EULER_INFO_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>EULER_AXIS_MALLOC</td>
<td>no memory euler angles and euler rotation axis information</td>
</tr>
<tr>
<td>EULER_IDF_DATA_MALLOC</td>
<td>no memory for array of pointers for the <em>idf_data</em> structures needed to hold the euler angle data read from the specified IDFS data source</td>
</tr>
<tr>
<td>EULER_BAD_SRC</td>
<td>the IDFS data source for the euler angle data is not a scalar instrument</td>
</tr>
<tr>
<td>EULER_TBL_MALLOC</td>
<td>no memory for table number / table operation information for euler angle data</td>
</tr>
<tr>
<td>BAD_EULER_FORMAT</td>
<td>the format specification field for the euler angle data is invalid</td>
</tr>
<tr>
<td>LESS_EULER_CONSTANTANGLES</td>
<td>the number of euler angle constants defined in the VIDF file is less than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>LESS_EULER_CONSTANT_AXIS</td>
<td>the number of euler rotation axis constants defined in the VIDF file is less than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>MORE_EULER_CONSTANTANGLES</td>
<td>the number of euler angle constants defined in the VIDF file is more than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>MORE_EULER_CONSTANT_AXIS</td>
<td>the number of euler rotation axis constants defined in the VIDF file is more than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>TOO_MANY_EULER</td>
<td>this data set defines more euler angles than the IDFS system can handle</td>
</tr>
<tr>
<td>CP_INFO_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>CP_INFO_IDF_MANY_BYTES</td>
<td>the number of element being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>CP_INFO_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>CP_INFO_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>CP_INFO_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>CP_TBL_MALLOC</td>
<td>no memory for table number / table operation information for celestial position angle data</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NO_CP_CONSTANT</td>
<td>the celestial position angle constants are not defined in the VIDF file</td>
</tr>
<tr>
<td>CP_STR_MALLOC</td>
<td>no memory for structure that holds celestial position angle information</td>
</tr>
<tr>
<td>CP_DATA_MALLOC</td>
<td>no memory for data and normalization factors for celestial position angle data</td>
</tr>
<tr>
<td>CP_BAD_SRC</td>
<td>the IDFS data source for the celestial position angle data is not a scalar instrument</td>
</tr>
<tr>
<td>NO_BKGD_CONSTANT</td>
<td>the background constants are not defined in the VIDF file</td>
</tr>
<tr>
<td>BKGD_TBL_MALLOC</td>
<td>no memory for table number / table operation information for background data</td>
</tr>
<tr>
<td>BKGD_BAD_SRC</td>
<td>the IDFS data source for the background data is not a scalar instrument</td>
</tr>
<tr>
<td>BKGD_MALLOC</td>
<td>no memory for structure that holds background information</td>
</tr>
<tr>
<td>BKGD_DATA_MALLOC</td>
<td>no memory for data and normalization factors for background data</td>
</tr>
<tr>
<td>BKGD_IDF_DATA_MALLOC</td>
<td>no memory for array of pointers for the idf_data structures needed to hold the</td>
</tr>
<tr>
<td></td>
<td>background data read from the specified IDFS data source</td>
</tr>
<tr>
<td>BKGD_INFO&gt;IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>BKGD_INFO_HEAD_MANY_BYTES</td>
<td>the number of element being requested is more than the number of elements available</td>
</tr>
<tr>
<td></td>
<td>for the selected field</td>
</tr>
<tr>
<td>BKGD_INFO_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>BKGD_INFO_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>BKGD_INFO_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>FILE_POS_INVALID_DATA</td>
<td>the data structure passed as an argument is not a valid data structure to use - may</td>
</tr>
<tr>
<td></td>
<td>have been previously freed</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>Error codes returned by read_drec ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by ReadVIDF ()</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

File_pos is the IDFS data positioning routine. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. This routine uses the currently opened files for the requested data set and sets the current data pointer to the data sample or sweep whose beginning time is closest to the requested start time. If btime_sec is equal to the value -1, the file position is set at the beginning of the current real-time data file. If btime_sec is set equal to the value -2, the file position is set to the present location within the current real-time data file. If the beginning time indicates post acquisition analysis, calls are made to the routine read_drec in order to position the data pointer as close to the requested start time as possible, using the btime_nano time component to get to the closest nanosecond, and the data structure is filled in and ready for access upon return from file_pos ()

Once the data set of interest has been successfully positioned, any ancillary data sources that are defined are also opened and positioned. If the data set of interest contains pitch angle information within the VIDF file, an attempt is made to open the IDFS data set that contains the magnetic field elements to be used in the pitch angle calculations and to set the data pointer for the pitch angle IDFS data set to the data sample or sweep whose beginning time is closest to the time at which the data set of interest has been positioned. If the data set of interest contains spacecraft potential information within the VIDF file, an attempt is made to open the IDFS data set that contains the spacecraft potential data values and to set the data pointer for the spacecraft potential IDFS data set to the data sample or sweep whose...
beginning time is closest to the time at which the data set of interest has been positioned. If the data set of interest contains start of spin information within the VIDF file, an attempt is made to open the IDFS data set that contains the time for each spin period and to set the data pointer for the start of spin IDFS data set to the data sample or sweep whose beginning time is closest to the time at which the data set of interest has been positioned. If the data set of interest contains euler angle information within the VIDF file, an attempt is made to open the IDFS data set that contains the euler angle and euler rotation axis information used in the euler angle calculations and to set the data pointer for the euler angle IDFS data set to the data sample or sweep whose beginning time is closest to the time at which the data set of interest has been positioned. If the data set of interest contains celestial position angle information within the VIDF file, an attempt is made to open the IDFS data set that contains the celestial position angle data (declination angle and right ascension angle) and to set the data pointer for the celestial position IDFS data set to the data sample or sweep whose beginning time is closest to the time at which the data set of interest has been positioned. If the data set of interest contains background information within the VIDF file, an attempt is made to open the IDFS data set that contains the background data values and to set the data pointer for the background IDFS data set to the data sample or sweep whose beginning time is closest to the time at which the data set of interest has been positioned.

Data positioning is performed only once for each unique parameter set. If additional calls are made to this routine with the same parameter set, the module simply returns the ALL_OKAY status code, with the exception being after a call to the module reset_experiment_info, which closes the existing IDFS data set and opens the next IDFS data set to be processed. It is imperative that a call to the file_pos routine be made immediately after a successful return from the reset_experiment_info module in order for the IDFS software to process the next IDFS data set correctly. Before the first call to the file_pos routine can be made, a call to the routine file_open with the identical data_key, exten and version parameters must have been made to obtain a set of file descriptors for the appropriate VIDF, header and data files.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter data_ptr is a pointer to the data structure that is to hold all data pertinent to the data set being processed. The data structure that is utilized is either an instance of the idf_data structure or the tensor_data structure. The data structure is created and the address to this structure is returned when a call to the create_idf_data_structure or create_tensor_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for
the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the `create_idf_data_structure` or `create_tensor_data_structure` routine N times to create N instances of the `idf_data` or `tensor_data` structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

**File_pos** is also the IDFS routine that allocates memory blocks which are used to hold and return information utilized by the IDFS routines. Memory is allocated to hold both the header and data record information. Whereas the size of the data records stay fixed, the size of the header records may change. **File_pos** allocates space based upon the size of the header record associated with the data record read. Memory expansion for the header information is handled by the `read_drec` routine. Memory is also allocated to hold information relevant to the application of the data calibration sets, to hold sensor time offset values and to hold full sweep values. The memory pointers for all of these elements are stored within an internally defined structure that is identified with a specific data set.

**File_pos** also allocates space to hold the calibration data and the instrument mode flags that are returned by the `read_drec` routine. If the currently assigned memory block is determined to be insufficient in size (too small), the memory block is dynamically expanded. The pointers to the two memory blocks are held in the `idf_data` structure. This data structure is described in section 1S of the IDFS Programmers Manual.

The last task of this routine is to allocate memory for structure(s) that hold the table values and critical status information for the sensors being processed. The memory pointer for this array of structures is stored within an internally defined structure that is identified with a specific data set. This routine assumes that the requested sensors have already been selected through the routine `select_sensor`. If the routine `select_sensor` is not called, all sensors are selected by the routine `file_open`.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable `USER_DATA`, which is set by the user, or in the user's home directory if the environment variable `USER_DATA` is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.
SEE ALSO
reset_experiment_info  1R
file_open  1R
read_drec  1R
select_sensor  1R
get_data_key  1R
get_version_number  1R
create_data_structure  1R
create_idf_data_structure  1R
create_tensor_data_structure  1R
libbase_idfs  1H
ret_codes  1H
idf_data  1S
tensor_data  1S

BUGS
None

EXAMPLES
Position the real-time IDFS data files associated with the virtual instrument RTLA, which is part of the RETE instrument/experiment, at the beginning of the data file. The RETE instrument/experiment is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}
```
status = file_open (data_key, ",", vnum, -1, -1, -1, 0, -1, -1, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_open routine.\n", status);
    exit (-1);
}

status = file_pos (data_key, ",", vnum, idf_data_ptr, -1, -1, -1, 0, -1, -1, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_pos routine.\n", status);
    exit (-1);
}
FIRST_IDFS_SENSOR

function – returns the first IDFS sensor that is defined within the sensor set for the current
data record

SYNOPSIS

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT first_idfs_sensor (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                          SDDAS_USHORT version, SDDAS_SHORT *sensor_num)

ARGUMENTS

data_key  - unique value which indicates the data set of interest
exten   - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version   - IDFS data set identification number which allows for multiple
               openings of the same data set
sensor_num  - sensor identification number
first_idfs_sensor - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for FIRST_IDFS_SENSOR

<table>
<thead>
<tr>
<th>STATUS_CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST_SEN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

First_idfs_sensor is the IDFS routine that will return the sensor identification number for
the first sensor that is defined within the sensor set of the current data record. The data set
of interest is referenced through the key value data_key which can be created using the
get_data_key module. This module is helpful when processing instrument status (mode)
data. Although the instrument status data is not sensor-specific, that is, the data pertain to
all sensors within the sensor set, the instrument status data is acquired using the IDFS read
routine read_drec and read_drec requires a sensor identification number as one of its arguments.

The parameter version allows multiple file openings for an IDFS data set. If the data,
header and VIFD file for the specified data set need to be opened just once for processing,
the same version number should be passed to all IDFS routines. However, for multiple file
openings, the version number should be unique and all file manipulations performed by the
IDFS routines will use the file descriptors defined for the version number specified. The
user should call the get_version_number routine to retrieve a unique version number
instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single
version number will suffice.
If the \texttt{file\_open} routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable \texttt{exten}. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER\_DATA, which is set by the user, or in the user's home directory if the environment variable USER\_DATA is not set. To open the default IDFS data files, \texttt{exten} should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set \texttt{exten} to a null string for real-time scenarios.

\subsection*{ERRORS}
All errors within this routine are returned through the status variable. The include file \texttt{ret\_codes.h}, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The \texttt{ret\_codes.h} file is described in section 1H of the IDFS Programmers Manual.

\subsection*{SEE ALSO}
\begin{itemize}
  \item file\_open \texttt{1R}
  \item read\_drec \texttt{1R}
  \item get\_data\_key \texttt{1R}
  \item get\_version\_number \texttt{1R}
  \item libbase\_idfs \texttt{1H}
  \item ret\_codes \texttt{1H}
\end{itemize}

\subsection*{BUGS}
None

\subsection*{EXAMPLES}
Retrieve the sensor identification number for the first sensor that is defined within the sensor set of the current data record.

\begin{verbatim}
#include "libbase_idfs.h"
#include "ret_codes.h"
SDDAS ULONG data_key;
SDDAS USHORT vnum;
SDDAS SHORT status, sensor_number;
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
  {
    printf ("\n Error %d returned by get\_data\_key routine.\n", status);
    exit (-1);
  }
get_version_number (&vnum);
\end{verbatim}
status = first_idfs_sensor (data_key, "", vnum, &sensor_number);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by first_idfs_sensor routine.\n", status);
    exit (-1);
}
FREE_experiment_info

function - frees all the memory allocated by the IDFS routines

SYNOPSIS

#include "libbase_idfs.h"

void free_experiment_info (void)

ARGUMENTS

No arguments for this routine

DESCRIPTION

Free_experiment_info is the IDFS routine that frees all memory that has been allocated by the IDFS routines. The computer operating system normally takes care of freeing any memory before terminating the program; however, for a clean exit, the user should call this module before exiting from the program. In addition, the user may call this module if a total restart of the IDFS software is desired without restarting the program. In the case of a total restart, the user is advised to call the module init_idfs before any other IDFS routine since the free_experiment_info routine merely frees allocated memory; it does not re-initialize variables used by the IDFS software.

If any idf_data structures were created using the create_idf_data_structure or create_data_structure routine, free_experiment_info will free the memory associated with elements contained in the idf_data structure and the data structure itself. The user must not free the memory since the IDFS software will also attempt to free the memory.

If any tensor_data structures were created using the create_tensor_data_structure or create_data_structure routine, free_experiment_info will free the memory associated with elements contained in the tensor_data structure and the data structure itself. The user must not free the memory since the IDFS software will also attempt to free the memory.

ERRORS

This routine returns no status or error codes.

SEE ALSO

init_idfs 1R
create_data_structure 1R
create_idf_data_structure 1R
create_tensor_data_structure 1R
libbase_idfs 1H

BUGS

None
EXAMPLES

The usage of this routine is quite simple since no parameters are needed:

```
#include "libbase_idfs.h"

free_experiment_info();
```
FREE_VERSION_INFO
   function - frees the memory allocated by the IDFS routines for the specified version number

SYNOPSIS
   #include "libbase_idfs.h"

   void free_version_info (SDDAS_USHORT version)

ARGUMENTS
   version  - IDFS data set identification number which allows for multiple
              openings of the same data set

DESCRIPTION
   Free_version_info is the IDFS routine that frees all memory that has been allocated by the
   IDFS routines for the specified version number. The computer operating system normally
   takes care of freeing any memory before terminating the program; however, for a clean exit,
   the user should call this module before exiting from the program. If the user desires a total
   restart of the IDFS software without restarting the program, the user should not use this
   module but should use the free_experiment_info module.

   The parameter version allows multiple file openings for an IDFS data set. If the data,
   header and VIDF file for the specified data set need to be opened just once for processing,
   the same version number should be passed to all IDFS routines. However, for multiple file
   openings, the version number should be unique and all file manipulations performed by the
   IDFS routines will use the file descriptors defined for the version number specified. The
   user should call the get_version_number routine to retrieve a unique version number
   instead of choosing this value themselves. The retrieval of multiple data parameters from a
   single data source does not constitute the need for multiple version numbers; a single
   version number will suffice.

ERRORS
   This routine returns no status or error codes.

SEE ALSO
   free_experiment_info  1R
   get_version_number    1R
   libbase_idfs          1H

BUGS
   None

EXAMPLE
   Free the memory allocated by the IDFS software for the specified version number. Assume
   this value has been previously set by the get_version_number routine.
#include "libbase_idfs.h"

SDDAS_USHORT vnum;

free_version_info (vnum);
GET_DATA_KEY

function - create a key value which identifies the data set of interest

SYNOPSIS

#include "libdb.h"
#include "ret_codes.h"

SDDAS_SHORT get_data_key (SDDAS_CHAR *pstr, SDDAS_CHAR *mstr,
                        SDDAS_CHAR *estr, SDDAS_CHAR *istr,
                        SDDAS_CHAR *vstr, SDDAS_ULONG *data_key)

ARGUMENTS

pstr  - the assigned database name for the project to be accessed
mstr  - the assigned database name for the mission to be accessed
estr  - the assigned database name for the experiment to be accessed
istr  - the assigned database name for the instrument to be accessed
vstr  - the assigned database name for the virtual instrument to be accessed
data_key - unique value which indicates the data set being requested
get_data_key - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for GET_DATA_KEY

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DKEY_PROJECT</td>
<td>invalid project name</td>
</tr>
<tr>
<td>DKEYMISSION</td>
<td>invalid mission name</td>
</tr>
<tr>
<td>DKEY_EXPERIMENT</td>
<td>invalid experiment name</td>
</tr>
<tr>
<td>DKEY_INSTRUMENT</td>
<td>invalid instrument name</td>
</tr>
<tr>
<td>DKEY_VINST</td>
<td>invalid virtual instrument name</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Get_data_key is the IDFS routine which creates a key that reflects the data set being accessed by utilizing the assigned database names for the project, mission, experiment, instrument and virtual instrument of interest. The IDFS routine fields_to_key performs the same function but works with the assigned database numbers instead of the assigned database names. Most of the IDFS routines utilize key values; therefore, a call to either this routine or to the fields_to_key routine must be made before any of the other IDFS routines that utilize a key value can be called.

The user selects the data set of interest by specifying the name of a virtual instrument from a specific instrument, which comes from a parent experiment within a mission which is associated with a specific project. All references for these items are through assigned database names. Since the IDFS data access software must interface with the database, the user must include the file libdb.h in their code when the get_data_key module is called.
ERRORS
All errors within this routine are returned through the status variable. The include file
\texttt{ret\_codes.h}, which includes all possible return values, should be included so that the
mnemonics for the return codes can be referenced. The \texttt{ret\_codes.h} file is described in
section 1H of the IDFS Programmers Manual.

SEE ALSO
fields\_to\_key \ 1R
ret\_codes \ 1H

BUGS
None

EXAMPLES
Retrieve the data key for the virtual instrument RTLA, which is part of the RETE
instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS
project.

\begin{verbatim}
#include "libdb.h"
#include "ret_codes.h"

SDDAS\_ULONG data\_key;
SDDAS\_SHORT ret\_val;

ret\_val = get\_data\_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data\_key);
if (ret\_val != ALL\_OKAY)
{
  printf ("\n Error %d returned by get\_data\_key routine.\n", ret\_val);
  exit (-1);
}
\end{verbatim}
GET_VERSION_NUMBER
function - returns a unique IDFS data set identification number

SYNOPSIS
#include "libbase_idfs.h"

void get_version_number (SDDAS_USHORT *version)

ARGUMENTS
version - IDFS data set identification number which allows for multiple
          openings of the same data set

DESCRIPTION
Get_version_number is the IDFS routine that returns a unique IDFS data set identification
number that is to be used as a parameter to the other IDFS routines. This parameter allows
multiple file openings for an IDFS data set. For multiple file openings of the same IDFS
data set, the version number must be unique and all file manipulations performed by the
IDFS routines will use the file descriptors defined for the version number specified. If the
user is opening many different IDFS data sets, but just opening each data set once, the user
may pass the same version number for each of the different IDFS data sets. For example, if
the user is going to process RTLA and RTLB data, one version number is sufficient. The
user should call the get_version_number routine to be guaranteed a unique version
number.

ERRORS
This routine returns no status or error codes.

BUGS
None

EXAMPLES
Retrieve a unique version number to be used by the IDFS routines.

#include "libbase_idfs.h"

SDDAS_USHORT vnum;

call_version_number (&vnum);
INIT_IDFS
  function - initializes the system for processing IDFS information

SYNOPSIS
  #include "libbase_idfs.h"

  void init_idfs (void)

ARGUMENTS
  No arguments for this routine

DESCRIPTION
  init_idfs is the IDFS routine that initializes the system to allow processing of the
  information contained in the IDFS files. A call must be made to this routine before any of
  the other IDFS routines documented in this manual can be utilized.

  Since the IDFS data access software must interface with the database, calls must be made to
  the dbInitialize and CfgInit modules when the init_idfs module is called. The user is
  referred to the webpages http://cluster/libdbSQL.html and http://cluster/libCfg.html for an
  explanation of these routines.

ERRORS
  This routine returns no status or error codes.

BUGS
  None

EXAMPLES
  The usage of this routine is quite simple since no parameters are needed:

    #include "libbase_idfs.h"

    CfgInit ();
    dbInitialize ();
    init_idfs ();
NEXT_FILE_START_TIME

function - returns the time that is to be used to retrieve the next data file

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT   next_file_start_time (SDDAS_ULONG data_key,
                        SDDAS_CHAR *exten, SDDAS_USHORT version,
                        SDDAS_CHAR mode_data, SDDAS_SHORT *start_yr,
                        SDDAS_SHORT *start_day, SDDAS_LONG *start_sec,
                        SDDAS_LONG *start_nano)

ARGUMENTS
  data_key  - unique value which indicates the data set of interest
  exten   - two character extension to be added to IDFS file names
             when default files are not to be used, otherwise a null string
  version   - IDFS data set identification number which allows for multiple
             openings of the same data set
  mode_data  - flag indicating if the time for instrument status (mode) data is
               being requested
               0 - the time for instrument status data is not being
               requested
               1 - the time for instrument status data is being
               requested
  start_yr  - year for retrieval of next data file
  start_day  - day of year for retrieval of next data file
  start_sec  - time of day in seconds for retrieval of next data file
  start_nano  - time of day residual in nanoseconds
  next_file_start_time - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for NEXT_FILE_START_TIME

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT_FILE_TIME_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>NEXT_FILE_TIME_FILE_OPEN</td>
<td>the user did not request mode data processing when file_open was called</td>
</tr>
<tr>
<td>NEXT_FILE_TIME_INFO_DUP</td>
<td>the requested data_key, exten, version combination has no memory allocated for the instrument status information</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Next_file_start_time is the IDFS routine that will return the start time that will trigger the retrieval of the next data file to be processed. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. This routine should be called only in the case of a playback database request. When the return code from the read_drec, read_tensor_data, start_image, fill_data, fill_discontinuous_data,
fill_mode_data, sweep_data, sweep_discontinuous_data, sweep_mode_data or file_pos routine indicates that the end of the current data file has been reached (LOS_STATUS or NEXT_FILE_STATUS), the next_file_start_time module should be called and the time values returned should be sent to the reset_experiment_info routine.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
reset_experiment_info 1R
get_version_number 1R
get_data_key 1R
file_pos 1R
read_drec 1R
read_tensor_data 1R
start_image 1R
fill_data 2R
fill_discontinuous_data 2R
fill_mode_data 2R
sweep_data 2R
sweep_discontinuous_data 2R
EXAMPLES

Determine the start time to be used to retrieve the next data file for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. In the calling sequence, indicate that the time for instrument status data is not being requested for the data set in question.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_LONG start_sec, start_nsec;
SDDAS_SHORT status, start_yr, start_day;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = next_file_start_time (data_key, ",", vnum, 0, &start_yr, &start_day,
    &start_sec, &start_nsec);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by next_file_start_time routine.\n", status);
    exit (-1);
}
```
override_potential_polynomial (1R)

OVERRIDE_POTENTIAL_POLYNOMIAL
function – overrides the slope and intercept values defined for the first order polynomial
that is used to adjust the spacecraft potential data used by the specified data set

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT override_potential_polynomial (SDDAS_ULONG data_key,
SDDAS_CHAR *exten, SDDAS_USHORT version,
SDDAS_FLOAT slope, SDDAS_FLOAT intercept)

ARGUMENTS
data_key    - unique value which indicates the data set of interest
exten     - two character extension to be added to IDFS
file names when default files are not to be used, otherwise a null string
version     - IDFS data set identification number which allows for multiple openings of the same data set
slope      - new slope value to be used in the polynomial equation
intercept    - new intercept value to be used in the polynomial equation
override_potential_polynomial - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for OVERRIDE_POTENTIAL_POLYNOMIAL

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERRIDE_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>OVERRIDE_NO_POT</td>
<td>there is no spacecraft potential data defined for this data set</td>
</tr>
<tr>
<td>OVERRIDE_NO_POT_TBLS</td>
<td>there are no tables defined for this data set which are a function of spacecraft potential data (tbl_var = 3)</td>
</tr>
<tr>
<td>OVERRIDE_TOO_MANY_POT_TBLS</td>
<td>there is more than one table defined for this data set which is a function of spacecraft potential data (tbl_var = 3)</td>
</tr>
<tr>
<td>OVERRIDE_TBL_FMT_MALLOC</td>
<td>no memory for the table format (tbl_fmt) values</td>
</tr>
<tr>
<td>OVERRIDE_BAD_TBL_FMT_VALUE</td>
<td>the table format values do not specify a first order polynomial</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>error codes returned by ReadVIDF ()</td>
</tr>
<tr>
<td></td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION
Override_potential_polynomial is the IDFS routine that can be used to override the coefficients of the polynomial equation that is used to modify the spacecraft potential data before it is used to convert the science data and / or scan data for the selected data set into scientific units. The original polynomial equation is defined within a table in the VIDF file
override_potential_polynomial (1R)

for the data set specified. This table must have the tbl_var value set to 3, indicating that the table is a function of spacecraft potential data. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. This routine must be called after the routine file_open and before the routine file_pos if this routine is to be utilized properly. If this routine is called multiple times with the identical data_key, exten and version parameters, the slope and intercept values from the last call will be saved and utilized.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_pos  1R
file_open  1R
get_data_key  1R
get_version_number  1R
ret_codes  1H
libbase_idfs  1H
convert_to_units  1R

BUGS
None
EXAMPLES
Modify the polynomial coefficients associated with the spacecraft potential data for the
virtual instrument CPXP1L, which is part of the 3DX1 instrument, which is part of the
PEACE experiment, which is part of the CLUSTER-1 mission, which is identified with the
CLUSTERII project.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_FLOAT slope, intercept;
SDDAS_ULONG data_key;
SDDAS_LONG btime_sec, btime_nano, etime_sec, etime_nano;
SDDAS_USHORT vnum;
SDDAS_SHORT status;

status = get_data_key ("CLUSTERII", "CLUSTER-1", "PEACE", "3DX1", "CPXP1L",
                      &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

btime_sec = (19 * 3600) + (50 * 60) + 0;
btime_nano = 0;
etime_sec = (19 * 3600) + (51 * 60) + 0;
etime_nano = 0;
status = file_open (data_key, "", vnum, 2002, 7, btime_sec, btime_nano, 2002, 7,
                    etime_sec, etime_nano, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d from file_open routine.\n", status);
    exit (-1);
}

slope = 1.0;
intercept = 1.5;
status = override_potential_polynomial (data_key, "", vnum, slope, intercept);
if (status != ALL_OKAY)
{
    printf ("\n Error %d from override_potential_polynomial routine.\n", status);
    exit (-1);
}
```
READ_DREC
function - read data from an IDFS file and return the data in raw units (telemetry level) in
the specified idf_data structure

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT read_drec (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                   SDDAS_USHORT version, void *idf_data_ptr,
                   SDDAS_SHORT sen, SDDAS_CHAR fwd, 
                   SDDAS_CHAR full_swp)

ARGUMENTS
data_key - unique value which indicates the data set of interest
exten  - two character extension to be added to IDFS file names when default
files are not to be used, otherwise a null string
version  - IDFS data set identification number which allows for multiple
openings of the same data set
idf_data_ptr - pointer to the idf_data structure that is to hold sensor data
and pertinent ancillary data for the data set of interest
sen  - sensor identification number
fwd - next time sample
   0 - do not advance to the next time sample
   1 - advance to the next time sample after obtaining data
full_swp - data return length (this option applicable only if the sensor
is associated with a scalar data set)
   0 - return a single data value
   1 - return n_sample data values
read_drec - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for READ_DREC

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DREC_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>DREC_NO_FILES</td>
<td>data and header files have not been opened</td>
</tr>
<tr>
<td>DREC_READ_ERROR</td>
<td>read error on data file</td>
</tr>
<tr>
<td>PARTIAL_READ</td>
<td>the number of bytes read from the file being accessed did not match the number of bytes requested. This code is returned only for the playback scenario. The code EOF_STATUS is returned for the real-time scenario.</td>
</tr>
<tr>
<td>DREC_HDR_READ_ERROR</td>
<td>read error on header file</td>
</tr>
<tr>
<td>DREC_HDR_MALLOC</td>
<td>no memory for header record information</td>
</tr>
<tr>
<td>DREC_HDR_REALLOC</td>
<td>no memory for header information expansion (header increased in size)</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RESET_DATA_MALLOC</td>
<td>no memory for sensor data array in the idf_data structure</td>
</tr>
<tr>
<td>RESET_DATA_REALLOC</td>
<td>no memory for sensor data array expansion in the idf_data structure</td>
</tr>
<tr>
<td>RESET_EULER_REALLOC</td>
<td>no memory for euler angle array expansion in the idf_data structure</td>
</tr>
<tr>
<td>RESET_ANGLE_REALLOC</td>
<td>no memory for azimuthal sample angle array expansion in the idf_data structure</td>
</tr>
<tr>
<td>RESET_PITCH_MALLOC</td>
<td>no memory for pitch angle array in the idf_data structure</td>
</tr>
<tr>
<td>RESET_PITCH_REALLOC</td>
<td>no memory for pitch angle array expansion in the idf_data structure</td>
</tr>
<tr>
<td>RESET_DCOS_MALLOC</td>
<td>no memory for direction cosine structure</td>
</tr>
<tr>
<td>RESET_DCOS_MALLOC</td>
<td>no memory for direction cosine values</td>
</tr>
<tr>
<td>RESET_DCOS_REALLOC</td>
<td>no memory for expansion of direction cosine values</td>
</tr>
<tr>
<td>RESET_MODE_REALLOC</td>
<td>no memory for expansion of instrument mode flags array</td>
</tr>
<tr>
<td>ALLOC_EV_REALLOC</td>
<td>no memory for sweep array expansion in the idf_data structure</td>
</tr>
<tr>
<td>RESET_CSET_MALLOC</td>
<td>no memory for calibration set size array in the idf_data structure</td>
</tr>
<tr>
<td>CRIT_TBL_NOT_FOUND</td>
<td>the table requested was not found amongst the sensor tables</td>
</tr>
<tr>
<td>PA_BAD_TIMES</td>
<td>the end time of the sample is less than the start time of the sample for pitch angle data</td>
</tr>
<tr>
<td>UPDATE_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>UPDATE_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>UPDATE_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>UPDATE_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>UPDATE_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>WRONG_HEADER_FORMAT</td>
<td>multi-dimensional IDFS data storage is not supported by this module</td>
</tr>
<tr>
<td>CREATE_TBL_MALLOC</td>
<td>no memory for table values (tbl) when the table is expanded using the coefficients from the VIDF file</td>
</tr>
<tr>
<td>CREATE_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>CREATE_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>CREATE_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>CREATE_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>CREATE_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>CREATE_BAD_TBL_OFFSET</td>
<td>invalid tbl_off value encountered</td>
</tr>
<tr>
<td>READ_IN_MALLOC</td>
<td>no memory for table values (tbl) read straight from the VIDF file</td>
</tr>
<tr>
<td>READ_IN_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>READ_IN_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>READ_IN_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>READ_IN_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>READ_IN_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>READ_IN_BAD_TBL_OFFSET</td>
<td>invalid tbl_off value encountered</td>
</tr>
<tr>
<td>NEW_SCALE_MALLOC</td>
<td>no memory to hold the scale factors to be applied to the table values</td>
</tr>
<tr>
<td>NEW_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>NEW_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>NEW_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>NEW_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>NEW_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>FILL_HEADER</td>
<td>the header record read is a fill header, indicating that the header record has not been received by the workstation at the time of the read from the file. This code is returned only for the playback scenario. The code EOF_STATUS is returned for the real-time scenario.</td>
</tr>
<tr>
<td>PA_BAD_FRAC</td>
<td>invalid normalization factor calculated for pitch angle data</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PA_BAD_SRC</td>
<td>the IDFS data source for the pitch angle data is not a scalar instrument</td>
</tr>
<tr>
<td>BAD_PA_FORMAT</td>
<td>the format specification field for the pitch angle data is invalid</td>
</tr>
<tr>
<td>NEW_BAD_TBL_OFFSET</td>
<td>invalid tbl_off value encountered</td>
</tr>
<tr>
<td>CHK_DATA_NOT_FOUND</td>
<td>an error was encountered when trying to access the structure that holds information</td>
</tr>
<tr>
<td></td>
<td>pertinent to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>NUM_CAL_REALLOC</td>
<td>no memory for expansion of calibration array</td>
</tr>
<tr>
<td>HDR_FMT_ONE_MALLOC</td>
<td>no memory for elements pertinent to original idfs definition</td>
</tr>
<tr>
<td>POT_BAD_FRAC</td>
<td>invalid normalization factor calculated for spacecraft potential data</td>
</tr>
<tr>
<td>POT_BAD_SRC</td>
<td>the IDFS data source for the spacecraft potential data is not a scalar instrument</td>
</tr>
<tr>
<td>BAD_SCPOT_FORMAT</td>
<td>the format specification field for the spacecraft potential data is invalid</td>
</tr>
<tr>
<td>POT_BAD_TIMES</td>
<td>the end time of the sample is less than the start time of the sample for spacecraft</td>
</tr>
<tr>
<td></td>
<td>potential data</td>
</tr>
<tr>
<td>RESET_POT_REALLOC</td>
<td>no memory for spacecraft potential array expansion in the idf_data structure</td>
</tr>
<tr>
<td>EULER_BAD_SRC</td>
<td>the IDFS data source for the euler angle data is not a scalar instrument</td>
</tr>
<tr>
<td>EULER_BAD_TIMES</td>
<td>the end time of the sample is less than the start time of the sample for euler angle</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td>BAD_EULER_FORMAT</td>
<td>invalid normalization factor calculated for euler angle data</td>
</tr>
<tr>
<td>LESS_EULER_CONSTANTSANGLES</td>
<td>the number of euler angle constants defined in the VIDF file is less than</td>
</tr>
<tr>
<td>LESS_EULER_CONSTANTSAXIS</td>
<td>the number of euler rotation axis constants defined in the VIDF file is less</td>
</tr>
<tr>
<td></td>
<td>than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>MORE_EULER_CONSTANTSANGLES</td>
<td>the number of euler angle constants defined in the VIDF file is more than</td>
</tr>
<tr>
<td>MORE_EULER_CONSTANTSAXIS</td>
<td>the number of euler rotation axis constants defined in the VIDF file is more</td>
</tr>
<tr>
<td></td>
<td>than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>CP_BAD_TIMES</td>
<td>the end time of the sample is less than the start time of the sample for celestial</td>
</tr>
<tr>
<td></td>
<td>position angle data</td>
</tr>
<tr>
<td>BAD_CP_FORMAT</td>
<td>the format specification field for the celestial position angle data is invalid</td>
</tr>
<tr>
<td>CP_BAD_FRAC</td>
<td>invalid normalization factor calculated for celestial position angle data</td>
</tr>
<tr>
<td>CP_BAD_SRC</td>
<td>the IDFS data source for the celestial position angle data is not a scalar instrument</td>
</tr>
<tr>
<td>RESET_CP_REALLOC</td>
<td>no memory for celestial position angle array expansion in the idf_data structure</td>
</tr>
<tr>
<td>BKGD_BAD_FRAC</td>
<td>invalid normalization factor calculated for background data</td>
</tr>
<tr>
<td>BKGD_BAD_TIMES</td>
<td>the end time of the sample is less than the start time of the sample for background</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td>BAD_BKGD_FORMAT</td>
<td>the format specification field for the background data is invalid</td>
</tr>
<tr>
<td>RESET_BKGD_REALLOC</td>
<td>no memory for background array expansion in the idf_data structure</td>
</tr>
<tr>
<td>RESET_TINFO_MALLOC</td>
<td>no memory for structure that holds coordinate transformation data in the idf_data</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data</td>
</tr>
<tr>
<td></td>
<td>being returned</td>
</tr>
<tr>
<td>DREC_EOF_NO_SENSOR</td>
<td>no data found for the requested sensor – eof on forward (real-time scenario only)</td>
</tr>
<tr>
<td>DREC_EOF_SENSOR</td>
<td>data found for the requested sensor – eof on forward (real-time scenario only)</td>
</tr>
<tr>
<td>DREC_NO_SENSOR</td>
<td>no data found for the requested sensor</td>
</tr>
<tr>
<td>EOF_STATUS</td>
<td>eof encountered on file being accessed (real-time scenario only)</td>
</tr>
<tr>
<td>LOS_STATUS</td>
<td>loss of signal encountered</td>
</tr>
</tbody>
</table>
**DESCRIPTION**

Read_drec is the IDFS data read routine, returning data for a single sensor. The sensor is indicated through the sensor number (sen). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The routine returns not only the data for the sensor but also most of the pertinent ancillary data concerning the state of the virtual instrument including time, instrument status (mode) values, all applicable correction and calibration data, sweep step values, azimuthal angle values, pitch angle values, spacecraft potential values and background values where applicable. All data is returned in raw units (telemetry format). To convert the data into the unit desired, the user should utilize the convert_to_units routine, which is explained in section 1R of the IDFS Programmers Manual.

In cases where pitch angle data is not needed, the routine turn_off_pitch_angle_computations may be called in order to save time performing unnecessary pitch angle computations. By default, euler angle information, if pertinent to the data set of interest, is not returned from the read_drec module unless the routine turn_on_euler_angle_computations has been called. In addition, celestial position angle information, if pertinent to the data set of interest, is not returned from the read_drec module unless the routine turn_on_celestial_position_computations has been called.

The returned data is placed in the idf_data structure that is referenced by the argument idf_data_ptr. The argument idf_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the create_idf_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_idf_data_structure routine N times to create N instances of the idf_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure. The contents of this structure is described in section 1S of the IDFS Programmers Manual.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the sensor returns scalar data, the number of data points returned in a given call can be changed through the variable full_swp. If full_swp is 0, the data is returned one sample at a time. If full_swp is 1, then all of the data, from the current data value through the end of the sensor set, are returned. With the exception of the first call to the read_rec routine, if full_swp is always set to 1 for a scalar sensor, the number of samples returned at each call is n_sample, which is defined in the header record. The number of samples returned at the first call depends on the initial position of the data pointer. The number of data values returned is always indicated in the idf_data structure.

After the data has been read, the current data pointer will be either be advanced to the next set of data values or remain at the current set of data values, depending upon the value of the variable fwd. For sweeping data, the pointer is advanced to the next full sweep with each forward. When used with scalar values, the pointer is advanced to the next value or to the next sensor set depending on whether the current value of full_swp is 0 or 1, respectively. By keeping the pointer at the same sensor set, repeated calls using the same virtual instrument but different sensors can be made, ensuring that all of the data returned was taken at the same time. Since the data is placed into the data arrays before the current data pointer is advanced, the user should check the status of the element filled_data within the idf_data structure if any status code other than ALL_OKAY is returned. This flag value will indicate if the data arrays have been filled and this data should be processed before further action is taken in accordance with the status code returned.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

There are two sets of time values returned by the read_drec routine. Within the idf_data structure, there are the elements byear, bday, bmilli, bnano, eyear, eday, emilli and enano. These time elements are associated with the requested sensor. The instrument status (mode) values are not sensor-specific, that is, they pertain to all sensors within the sensor set. Therefore, the time span encompassed by the instrument status values is specified in the elements mode_byear, mode_bday, mode_bmilli, mode_bnano, mode_eyear, mode_eday, mode_emilli and mode_enano within the idf_data structure.
ERRORS
All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
- reset_experiment_info 1R
- file_pos 1R
- convert_to_units 1R
- create_data_structure 1R
- create_idf_data_structure 1R
- get_data_key 1R
- get_version_number 1R
- turn_off_pitch_angle_computations 1R
- turn_on_euler_angle_computations 1R
- turn_on_celestial_position_computations 1R
- ret_codes 1H
- libbase_idfs 1H
- idf_data 1S

BUGS
None

EXAMPLES
Obtain one sweep of data from sensor 2 in the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. Since this is a sweeping instrument, the full_swp variable is set to 1 (can not retrieve part of a sweep). The pointer is moved to the next full sweep in the data array after the data is obtained. The data is returned in the idf_data structure referenced by the pointer idf_data_ptr.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"
SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);
```
status = create_idf_data_structure(&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit(-1);
}
status = read_drec(data_key, "", vnum, idf_data_ptr, 2, 1, 1);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by read_drec routine.\n", status);
    exit(-1);
}
READ_DREC_SPIN
function - read data from an IDFS file and return data for a complete spin in raw units (telemetry level) for the sensor requested

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT  read_drec_spin (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                                 SDDAS_USHORT version, SDDAS_SHORT sen,
                                 SDDAS_USHORT *start_ele,  SDDAS_FLOAT *start_frac,
                                 SDDAS_USHORT *stop_ele,  SDDAS_FLOAT *stop_frac,
                                 SDDAS_LONG *num_sweeps,  void ***data_ptrs)

ARGUMENTS
  data_key  - unique value which indicates the data set of interest
  exten   - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
  version   - IDFS data set identification number which allows for multiple
openings of the same data set
  sen   - sensor identification number
  start_ele  -  element number within the sweep where the spin starts
  start_frac  - percentage of data that is to be included for the element
within the sweep where the spin starts
  stop_ele  -  element number within the sweep where the spin stops
  stop_frac  - percentage of data that is to be included for the element
within the sweep where the spin stops
  num_sweeps  - the number of sweeps processed for the spin
  data_ptrs  - array of pointers to the idf_data structure(s) that hold sensor
data and pertinent ancillary data for the data set of interest for
each sweep processed for the spin
  read_drec_spin - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for READ_DREC_SPIN

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_SPIN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>READ_SPIN_NO_START</td>
<td>user did not call start_of_spin for this combination before calling this module</td>
</tr>
<tr>
<td>READ_SPIN_SENSOR_NOT_FOUND</td>
<td>the requested data_key, exten, version combination for one of the sensors being processed has no memory allocated for processing</td>
</tr>
<tr>
<td>READ_SPIN_DSRC_READ</td>
<td>error reading next record from spin data source to get next spin period</td>
</tr>
<tr>
<td>READ_SPIN_DSRC_BACK_SPIN</td>
<td>the next spin period went backwards in time</td>
</tr>
<tr>
<td>START_ELE_BAD_SENSOR</td>
<td>the sensor being requested is an invalid sensor number</td>
</tr>
<tr>
<td>START_ELE_SPIN_NO_SENSOR</td>
<td>the sensor being requested was not selected as a sensor to be processed for the data set in question (user did not call select_sensor for this combination)</td>
</tr>
</tbody>
</table>

147                                                                                     December 28, 2012
### STATUS CODE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRONG_HEADER_FORMAT</td>
<td>multi-dimensional IDFS data storage is not supported by this module</td>
</tr>
<tr>
<td>READ_SPIN_ALL_REALLOC</td>
<td>no memory for expansion of the array of pointers to the idf_data structures that are allocated to hold the data for each sweep within the spin</td>
</tr>
<tr>
<td>READ_SPIN_PARTIAL</td>
<td>a partial spin is being returned since there is no further data available past this point in the spin</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data being returned</td>
</tr>
<tr>
<td>READ_SPIN_DATA_GAP</td>
<td>a partial spin is being returned since a data gap was encountered while acquiring the current spin</td>
</tr>
<tr>
<td>READ_SPIN_TERMINATE</td>
<td>processing must be terminated since data for start of spin data source is no longer available – partial spin is returned</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>data for requested sensor is returned</td>
</tr>
</tbody>
</table>

### DESCRIPTION

**Read_drec_spin** is the IDFS data read routine that returns a full spin of data for a single sensor. The sensor is indicated through the sensor number (sen). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. **Read_drec_spin** make use of the read_drec routine and therefore, returns not only the data for the sensor but also most of the pertinent ancillary data concerning the state of the virtual instrument including time, instrument status (mode) values, all applicable correction and calibration data, sweep step values, azimuthal angle values, pitch angle values, spacecraft potential values and background values where applicable.

The argument data_ptrs is an array of pointers to the collection of idf_data structures that hold all data pertinent to the spin being processed. There is basically one idf_data structure allocated for each sweep within the spin. The total number of sweeps contained within the spin being processed is returned in the argument num_sweeps. The structures are created as the need for another idf_data structure is encountered and the address to the newly created structure is added to the array of pointers referenced by the data_ptrs argument. These data structures are re-used as successive spins are processed; therefore, the user must extract all data that is returned prior to the next call to the read_drec_spin routine. The contents of the idf_data structure is described in section 1S of the IDFS Programmers Manual. All data is returned in raw units (telemetry format). To convert the data into the unit desired, the user should utilize the convert_to_units routine, which is explained in section 1R of the IDFS Programmers Manual.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number.
instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single
version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of
IDFS files, the two character extension applied to these data files must be supplied to this
routine within the string variable exten. These files must have the identical name as the
IDFS files with the two character identification code appended to the end of the file names
(i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files
must reside either in the directory specified by the environment variable USER_DATA,
which is set by the user, or in the user's home directory if the environment variable
USER_DATA is not set. To open the default IDFS data files, exten should be set to a null
string. The usage of modified data sets is limited to post acquisition data; therefore, it is
suggested that the user set exten to a null string for real-time scenarios.

For sweeping data, the start of each spin period does not always correlate with the first
element of the sweep. The start_ele argument returns the element number at which the spin
begins for the first sweep returned for the spin being processed. Likewise, the stop_ele
argument returns the element number at which the spin ends for the last sweep returned for
the spin being processed.

Within the IDFS paradigm, there are two methods utilized to determine the start of spin for
an IDFS data source. The first method is referred to as the angular method since the start of
spin is flagged as the point at which the azimuthal angle crosses over at 0 degrees, taking
into account some tolerance factor. The second method allows the time of each spin to be
explicitly defined and is specified by defining an IDFS data source that is to be used to
determine the spin periods within the VIDF for the data set of interest. When the second
method is utilized, the arguments start_frac and stop_frac will return values between 0.0
and 1.0, indicating the percentage of data that is to be included for the elements within the
sweep where the spin starts (start_ele) and stops (stop_ele). This percentage is calculated
based upon the start / stop time for the spin period and the start / stop time for the first and
last sweep processed. When the angular method is utilized, the arguments start_frac and
stop_frac will be set at 1.0 since the angle value pertains to the entire duration of the step.
The user does not need to concern themselves with which method is utilized and it is up to
their discretion whether they wish to utilize the contents of these two arguments when
processing the data returned by the read_drec_spin module.

Unlike the read_drec and read_tensor_data modules, the read_drec_spin module
automatically handles the acquisition of the next data file when the end of the current data
file has been reached. This action is performed since this module tries to retrieve a full spin
of data, not just a single sweep of data. Since the acquisition is automatically performed,
there is no need for the user to call the reset_experiment_info module within their code.

ERRORS
All errors within this routine are returned through the status variable. The include file
ret_codes.h, which includes all possible return values, should be included so that the
mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO

file_open 1R
file_pos 1R
convert_to_units 1R
start_of_spin 1R
get_data_key 1R
get_version_number 1R
read_drec 1R
ret_codes 1H
libbase_idfs 1H
idf_data 1S

BUGS
None

EXAMPLES

Obtain one spin of data from sensor 0 in the virtual instrument CP3DRH, which is part of the 3DR instrument, which is part of the PEACE experiment, which is part of the CLUSTER-2 mission, which is identified with the CLUSTERII project.

#include "libbase_idfs.h"
#include "ret_codes.h"

struct idf_data *EXP_DATA;
register SDDAS_USHORT  k;
register SDDAS_LONG swp_num;
SDDAS_ULONG data_key;
SDDAS_LONG num_sweeps, last_sweep, *tbl_oper;
SDDAS_FLOAT conv_data[1000], start_frac, stop_frac;
SDDAS_USHORT vnum, start_ele, stop_ele, start_index, stop_index;
SDDAS_SHORT status, rcode;
SDDAS_CHAR *tbls_to_apply, num_tbls;
void **data_arrays, *idf_data_ptr;

status = get_data_key("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH",
&data_key);
if (status != ALL_OKAY)
{
    printf("\nError %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);
Retrieve the raw units for the data.

num_tbls = 0;
tbls_to_apply = NULL;
tbl_oper = NULL;

status = read_drec_spin(data_key, "", vnum, 0, &start_ele, &start_frac, &stop_ele, &stop_frac, &num_sweeps, &data_arrays);

if (status == ALL_OKAY || status == READ_SPIN_TERMINATE || status == READ_SPIN_PARTIAL || status == READ_SPIN_DATA_GAP)
{
    last_sweep = num_sweeps – 1;
    for (swp_num = 0; swp_num < num_sweeps; ++swp_num)
    {
        idf_data_ptr = *(data_arrays + swp_num);
        EXP_DATA = (struct idf_data *) idf_data_ptr;
        printf ("START_TIME YEAR = %4d START_TIME DAY = %03d
                START TIME_MS = %ld START TIME_NS = %ld",
                EXP_DATA->byear, EXP_DATA->bday, EXP_DATA->bmilli,
                EXP_DATA->bnano);
        printf ("END_TIME YEAR = %4d END_TIME DAY = %03d
                END TIME_MS = %ld END TIME_NS = %ld",
                EXP_DATA->eyear, EXP_DATA->eday, EXP_DATA->emilli,
                EXP_DATA->enano);
        rcode = convert_to_units(data_key, "", vnum, idf_data_ptr, 0, SENSOR, 0,
                num_tbls, tbls_to_apply, tbl_oper, conv_data, 0, 0);
        if (rcode != ALL_OKAY)
        {
            printf ("Error %d from convert_to_units.\n", rcode);
            exit (-1);
        }
    }
}
if (swp_num == 0)
{
    start_index = start_ele;
    stop_index = EXP_DATA->num_sample;
    conv_data[start_ele] *= start_frac;
}
else if (swp_num == last_sweep)
{
    start_index = 0;
    stop_index = stop_ele;
    conv_data[stop_ele] *= stop_frac;
}
else
{
    start_index = 0;
    stop_index = EXP_DATA->num_sample;
}

/**************************************************************/
/*  Print data values, 6 values per row, in exponential format.                           */
/**************************************************************/

for (k = start_index; k < stop_index; ++k)
{
    if (k % 6 == 0)
        printf ("\n");
    printf ("%.6f ", conv_data[k]);
}

printf ("\n\n");
READ_TENSOR_DATA
function - read data from a multi-dimensional IDFS data set and return the data in raw units (telemetry level) in the specified tensor_data structure

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT read_tensor_data (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, void *tensor_data_ptr,
SDDAS_SHORT sen, SDDAS_CHAR fwd)

ARGUMENTS
data_key - unique value which indicates the multi-dimensional IDFS data set of interest
exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple openings of the same data set
tensor_data_ptr - pointer to the tensor_data structure that is to hold sensor data and pertinent ancillary data for the data set of interest
sen - sensor identification number
fwd - next time sample
  0 - do not advance to the next time sample after obtaining data
  1 - advance to the next time sample after obtaining data

read_tensor_data - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for READ_TENSOR_DATA

<table>
<thead>
<tr>
<th>STATUS_CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TENSOR_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>WRONG_HEADER_FORMAT</td>
<td>this module should only be called for single-valued multi-dimensional IDFS data</td>
</tr>
<tr>
<td>TENSOR_NO_FILES</td>
<td>data and header files have not been opened</td>
</tr>
<tr>
<td>TENSOR_READ_ERROR</td>
<td>read error on data file for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>PARTIAL_READ</td>
<td>the number of bytes read from the file being accessed did not match the number of bytes requested. This code is returned only for the playback scenario. The code EOF_STATUS is returned for the real-time scenario.</td>
</tr>
<tr>
<td>FILL_HEADER</td>
<td>the header record read is a fill header, indicating that the header record has not been received by the workstation at the time of the read from the file. This code is returned only for the playback scenario. The code EOF_STATUS is returned for the real-time scenario.</td>
</tr>
<tr>
<td>TENSOR_HDR_READ_ERROR</td>
<td>read error on header file for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>TENSOR_HDR_MALLOC</td>
<td>no memory for header record information for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TENSOR_HDR_REALLOC</td>
<td>no memory for header information expansion for multi-dimensional IDFS data (header increased in size)</td>
</tr>
<tr>
<td>TENSOR_DATA_MALLOC</td>
<td>no memory for sensor data array in the tensor_data structure</td>
</tr>
<tr>
<td>TENSOR_DATA_REALLOC</td>
<td>no memory for sensor data array expansion in the tensor_data structure</td>
</tr>
<tr>
<td>TENSOR_MODE_MALLOC</td>
<td>no memory for instrument mode flags array returned in the tensor_data structure</td>
</tr>
<tr>
<td>TENSOR_DATA_REALLOC</td>
<td>no memory for expansion of instrument mode flags array</td>
</tr>
<tr>
<td>TENSOR_DATA_TDW_LEN</td>
<td>only byte-oriented multi-dimensional IDFS data can be defined</td>
</tr>
<tr>
<td>UPDATE_IDF_NO_FILL</td>
<td>a fill value must be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_PA_DEF</td>
<td>pitch angle can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_POT_DEF</td>
<td>spacecraft potential data can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_SPIN_DEF</td>
<td>start of spin data source can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_PMI_DEF</td>
<td>euler angle can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_BKGD_DEF</td>
<td>background data can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_BAD_CP_DEF</td>
<td>celestial position angles can not be specified for multi-dimensional IDFS data</td>
</tr>
<tr>
<td>UPDATE_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>UPDATE_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>UPDATE_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>UPDATE_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>UPDATE_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>HDR_FMT_TWO_MALLOC</td>
<td>no memory for elements pertinent to multi-dimensional IDFS definition</td>
</tr>
<tr>
<td>HDR_FMT_TWO_DQUAL</td>
<td>the size of the data quality tensor does not match the size of the multi-dimensional IDFS data set</td>
</tr>
<tr>
<td>TENSOR_DQUAL_MALLOC</td>
<td>no memory for data quality values returned in the tensor_data structure</td>
</tr>
<tr>
<td>TENSOR_DQUAL_REALLOC</td>
<td>no memory for expansion of data quality values in the tensor_data structure</td>
</tr>
<tr>
<td>CRIT_ACT_MALLOC</td>
<td>no memory for critical action information</td>
</tr>
<tr>
<td>LESS_EULER_CONSTANTANGLES</td>
<td>the number of euler angle constants defined in the VIDF file is less than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>LESS_EULER_CONSTANT_AXIS</td>
<td>the number of euler rotation axis constants defined in the VIDF file is less than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>MORE_EULER_CONSTANTANGLES</td>
<td>the number of euler angle constants defined in the VIDF file is more than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>MORE_EULER_CONSTANT_AXIS</td>
<td>the number of euler rotation axis constants defined in the VIDF file is more than the number of euler angles defined in the VIDF file</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data being returned</td>
</tr>
<tr>
<td>TENSOR_NO_SENSOR</td>
<td>no data found for the requested sensor</td>
</tr>
<tr>
<td>TENSOR_EOF_NO_SENSOR</td>
<td>no multi-dimensional IDFS data found for the requested sensor – eof on forward (real-time scenario only)</td>
</tr>
<tr>
<td>TENSOR_EOF_SENSOR</td>
<td>multi-dimensional IDFS data found for the requested sensor – eof on forward (real-time scenario only)</td>
</tr>
<tr>
<td>NEXT_FILE_STATUS</td>
<td>the end of the current data file being processed has been reached</td>
</tr>
<tr>
<td>LOS_STATUS</td>
<td>loss of signal encountered</td>
</tr>
<tr>
<td>EOF_STATUS</td>
<td>eof encountered on file being accessed (real-time scenario only)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>single-valued multi-dimensional IDFS data for requested sensor is returned</td>
</tr>
</tbody>
</table>
DESCRIPTION

Read_tensor_data is the multi-dimensional IDFS data read routine used to retrieve single-valued tensor data for a single sensor. The sensor is indicated through the sensor number (sen). The multi-dimensional IDFS data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The routine returns not only the single-valued multi-dimensional IDFS data for the sensor but also most of the pertinent ancillary data concerning the state of the virtual instrument including time and instrument status (mode) values where applicable. For the time being, multi-dimensional IDFS data can not be dynamically converted to any other physical unit; therefore, the data must be stored in the physical unit desired when the data set is created. However, the read_tensor_data module will return two sets of data within the tensor_data structure. One set represents the raw integer values that are stored within the data record and one set represents the floating point values that result when transferring the raw integer values into the data type defined by d_type in the VIDF file for the IDFS data set being processed.

The returned data is placed in the tensor_data structure that is referenced by the argument tensor_data_ptr. The argument tensor_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the create_tensor_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct single-valued multi-dimensional IDFS data sets. However, if more than one structure is needed, the user may call the create_tensor_data_structure routine N times to create N instances of the tensor_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure. The contents of this structure is described in section 1S of the IDFS Programmers Manual.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable
read_tensor_data (1R)

USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

After the data has been read, the current data pointer will be either be advanced to the next set of data values or remain at the current set of data values, depending upon the value of the variable fwd. By keeping the pointer at the current set of data values, repeated calls using the same virtual instrument but different sensors can be made, ensuring that all of the data returned was taken at the same time. Since the data is placed into the data arrays before the current data pointer is advanced, the user should check the status of the element filled_data within the tensor_data structure if any status code other than ALL_OKAY is returned. This flag value will indicate if the data arrays have been filled and this data should be processed before further action is taken in accordance with the status code returned.

There are two sets of time values returned by the read_tensor_data routine. Within the tensor_data structure, there are the elements byear, bday, bmilli, bnano, eyear, eday, emilli and enano. These time elements are associated with the data for the requested sensor. The instrument status (mode) values are not sensor-specific, that is, they pertain to all sensors within the data record being processed. Therefore, the time span encompassed by the instrument status values is specified in the elements mode_byear, mode_bday, mode_bmilli, mode_bnano, mode_eyear, mode_eday, mode_emilli and mode_enano within the tensor_data structure.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
get_data_key 1R
get_version_number 1R
create_data_structure 1R
create_tensor_data_structure 1R
reset_experiment_info 1R
ret_codes 1H
libbase_idfs 1H
tensor_data 1S

BUGS
None

December 28, 2012
EXAMPLES

Obtain multi-dimensional IDFS data for sensor 0 from the virtual instrument NPD1BM16, which is part of the NPD instrument, which is part of the ASPERA-3 experiment, which is part of the Mars_Express mission, which is identified with the MARS project. The pointer is moved to the next time sample in the data array after the data is obtained. The data is returned in the tensor_data structure referenced by the pointer tensor_data_ptr.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
void *tensor_data_ptr;
status = get_data_key ("MARS", "Mars_Express", "ASPERA-3", "NPD", "NPD1BM16",
&data_key);
if (status != ALL_OKAY)
{
    printf ("Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_tensor_data_structure (&tensor_data_ptr);
if (status != ALL_OKAY)
{
    printf ("Error %d returned by create_tensor_data_structure routine.\n", status);
    exit (-1);
}

status = read_tensor_data (data_key, ",", vnum, tensor_data_ptr, 0, 1);
if (status != ALL_OKAY)
{
    printf ("Error %d returned by read_tensor_data routine.\n", status);
    exit (-1);
}
```

157  December 28, 2012
READ_IDF
function – retrieve information from the VIDF file for the specified IDFS data set

DESCRIPTION
This module has been moved into a separately maintained library. The user is referred to the webpage http://cluster/libVIDF.html for an explanation of the interface to this module. The IDFS data access software makes use of both the read_idf and ReadVIDF modules. The examples provided at the beginning of this manual make use of only the read_idf module.
RESET_EXPERIMENT_INFO

function - closes the current data files and opens the next set of data files to be processed

SYNOPSIS

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT reset_experiment_info (SDDAS_ULONG data_key,
SDDAS_CHAR *exten, SDDAS_USHORT version,
SDDAS_SHORT btime_yr, SDDAS_SHORT btime_day,
SDDAS_LONG btime_sec, SDDAS_SHORT btime_nano,
SDDAS_SHORT etime_yr, SDDAS_SHORT etime_day,
SDDAS_LONG etime_sec, SDDAS_LONG etime_nano)

ARGUMENTS

data_key   - unique value which indicates the data set of interest
exten    - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version    - IDFS data set identification number which allows for multiple openings of the same data set
btime_yr   - beginning year for data being requested
btime_day   - beginning day of year for data being requested
btime_sec   - beginning time of day in seconds for data being requested
btime_nano   - beginning time of day residual in nanoseconds
etime_yr   - ending year for data being requested
etime_day   - ending day of year for data being requested
etime_sec   - ending time of day in seconds for data being requested
etime_nano   - ending time of day residual in nanoseconds
reset_experiment_info - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for RESET_EXPERIMENT_INFO

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>Error codes returned by file_open ()</td>
</tr>
<tr>
<td></td>
<td>all data files opened</td>
</tr>
</tbody>
</table>

In addition to the status codes listed above, other error/status codes may be returned in the case of a database request. The user is referred to the webpage http://cluster/libdbSQL.html for an explanation of the interface to the database which is used by the IDFS data access software. The write-up for the modules dbIDFSGetRealTimeFile and dbIDFSGetFile are pertinent to the reset_experiment_info routine.
DESCRIPTION

Reset_experiment_info is the IDFS routine that may be used when the end of the current data file has been reached and the return code from the read_drec, read_tensor_data, start_image, fill_data, fill_discontinuous_data, fill_mode_data, sweep_data, sweep_discontinuous_data, sweep_mode_data or file_pos routine indicates that more data files need to be processed (NEXT_FILE_STATUS). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. This routine closes the currently opened files for the requested data set (data_key), file name extension (exten) and IDFS data set identification number (version), memory arrays that were allocated based upon VIDF information are freed and the next set of data files are opened. If the data set of interest contains pitch angle information, the IDFS data access software will automatically take care of data file management for the pitch angle IDFS data set. The IDFS data access software performs the same tasks when the data set of interest contains spacecraft potential information, background information and / or start of spin information.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The beginning time that is passed to this routine is dependent upon whether playback or real-time processing is desired. If real-time data files are being utilized, the value for btime_sec should be set to -1 so that the file positioning routine file_pos will position the file pointer at the beginning of the new real-time data file. If playback data files are being utilized, the routine next_file_start_time should be called to retrieve the start time that will trigger the retrieval of the next data file to be processed. The time values that are passed into this routine are used to make an internal call to the file_open routine in order to retrieve the next set of data files. It is imperative that a call to the file_pos routine be made immediately after a successful return from the reset_experiment_info module in order for the IDFS software to process the next IDFS data set correctly.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null
string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
file_pos 1R
read_drec 1R
read_tensor_data 1R
start_image 1R
fill_data 2R
fill_mode_data 2R
fill_discontinuous_data 2R
sweep_data 2R
sweep_mode_data 2R
sweep_discontinuous_data 2R
next_file_start_time 1R
get_data_key 1R
get_version_number 1R
ret_codes 1H
libbase_idfs 1H
idf_data 1S
tensor_data 1S

BUGS
None

EXAMPLES
The end of the current playback data file has been reached and more data exists for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. Retrieve the next set of data files. External variables are utilized to illustrate that the user-requested end time is set before this code is executed.

#include "libbase_idfs.h"
#include "ret_codes.h"

extern SDDAS_SHORT etime_yr, etime_day;
extern SDDAS_LONG etime_sec, etime_nano;
SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_LONG btime_sec, btime_nano;
SDDAS_SHORT status, btime_yr, btime_day;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = next_file_start_time (data_key, "", vnum, 0, &btime_yr, &btime_day,
                                &btime_sec, &btime_nano);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by next_file_start_time routine.\n", status);
    exit (-1);
}

status = reset_experiment_info (data_key, "", vnum, btime_yr, btime_day,
                                btime_sec, btime_nano, etime_yr, etime_day,
                                etime_sec, etime_nano);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by reset_experiment_info routine.\n", status);
    exit (-1);
}
**SELECT_SENSOR**

function - indicate which sensors are to be processed for the data set specified

**SYNOPSIS**

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT select_sensor (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                           SDDAS_USHORT version, SDDAS_SHORT sensor)
```

**ARGUMENTS**

- data_key: unique value which indicates the data set of interest
- exten: two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- version: IDFS data set identification number which allows for multiple openings of the same data set
- sensor: sensor identification number
- select_sensor: routine status (see TABLE 1)

**TABLE 1. Status Codes Returned for SELECT_SENSOR**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEL_SEN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

**Select_sensor** is the IDFS sensor selection routine. The data set of interest is referenced through the key value **data_key** which can be created using the **get_data_key** module. This routine does not have to be utilized by the programmer since the routine **file_open** sets the internal flags to indicate that all sensors associated with the data set are to be processed and memory to hold information concerning each sensor is to be allocated by the routine **file_pos**. However, in order to conserve space and to avoid unnecessary processing of sensors which are not going to be utilized by the programmer, this routine may be called once for each desired sensor. When this routine is called, the internal flags are reset such that only the requested sensors will be processed and have space allocated. This routine **must** be called before the routine **file_pos** if this routine is to be utilized properly.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the **get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

Within the IDFS paradigm, an instrument may be classified as a vector instrument or as a scalar instrument when dealing with data that is not stored in multi-dimensional IDFS format. A vector instrument is an instrument whose sensors represent multi-value data sets as opposed to a scalar instrument whose sensors represent a set of singular data values. If the data set to be processed is a vector instrument, the user should not call this routine if the `center_and_band_values` routine is to be utilized. If this routine is called, erroneous center sweep and/or band width values may be computed.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_pos` 1R
- `file_open` 1R
- `get_data_key` 1R
- `get_version_number` 1R
- `center_and_band_values` 2R
- `ret_codes` 1H
- `libbase_idfs` 1H

**BUGS**

None

**EXAMPLES**

The RTLA virtual instrument, which is part of the RETE instrument/experiment, has 5 sensors associated with it, referenced as sensors 0, 1, 2, 3, and 4. The RETE instrument/experiment is part of the TSS-1 mission, which is identified with the TSS project. Select sensors 0 and 3 to be the only two sensors to be processed and to have space allocated.
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTL", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);

status = select_sensor (data_key, "", vnum, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by select_sensor routine.\n", status);
    exit (-1);
}

status = select_sensor (data_key, "", vnum, 3);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by select_sensor routine.\n", status);
    exit (-1);
}
START_IMAGE
function - positions the file pointers at the beginning of an image

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT start_image (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, void *idf_data_ptr)

ARGUMENTS
  data_key  - unique value which indicates the data set of interest
  exten   - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
  version   - IDFS data set identification number which allows for multiple
openings of the same data set
  idf_data_ptr  - pointer to the idf_data structure that is to hold sensor
data and pertinent ancillary data for the data set of interest
  start_image  - routine status (see TABLE 1)

TABLE 1. Status Code Returned for START_IMAGE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>IMAGE_READ_ERROR</td>
<td>read error on data or header file</td>
</tr>
<tr>
<td>IMAGE_HDR_MALLOC</td>
<td>no memory for header record information</td>
</tr>
<tr>
<td>IMAGE_HDR_REALLOC</td>
<td>no memory for header information expansion (header increased in size)</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data being returned</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>Error codes returned by read_drec ()</td>
</tr>
</tbody>
</table>

DESCRIPTION
Start_image is the IDFS routine that positions the file descriptors at the start of an image. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. In order to store an image within the IDFS paradigm, at least one calibration set must be defined, calibration set 0, to hold the scan line number and this is a single quantity (one value, not an array of values). Second, the scan line value must increase from sensor set to sensor set within a single data record. This routine uses the currently opened files for the requested data set and sets the current data pointer to the data sample or sweep whose scan line calibration value is set to zero. This routine must be called AFTER a call to the file_pos routine has been made since the file_pos routine sets the data pointer to the data sample or sweep whose beginning time is closest to that requested by the user. The start_image routine assumes that the file_pos routine has been already been called. Data positioning is performed only once for each unique parameter set. If additional calls are made to this routine with the same parameter set, the module simply
returns the **ALL_OKAY** status code, with the exception being after a call to the module `reset_experiment_info`, which closes the existing IDFS data set and opens the next IDFS data set to be processed.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the **get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter **idf_data_ptr** is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the `create_idf_data_structure` routine is made. The user also has the option of calling the module `create_data_structure`, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the `create_idf_data_structure` routine N times to create N instances of the **idf_data** structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the **file_open** routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable **exten**. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ix). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, **exten** should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set **exten** to a null string for real-time scenarios.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open`  
- `file_pos`  
- `read_drec`  
- `reset_experiment_info`
EXAMPLES
Position the default IDFS data files associated with the virtual instrument SAIA, which is part of the SAI instrument/experiment, at the beginning of the data file and at the start of the image. The SAI instrument/experiment is part of the DE-1 mission, which is identified with the DE (Dynamics Explorer) project.

```
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
void *idf_data_ptr;

status = get_data_key ("DE", "DE-1", "SAI", "SAI", "SAIA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = file_pos (data_key, ",", vnum, idf_data_ptr, -1, -1, -1, 0, -1, -1, -1, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_pos routine.\n", status);
    exit (-1);
}
```
status = start_image (data_key, ",", vnum, idf_data_ptr);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by start_image routine.\n", status);
    exit (-1);
}
START_OF_SPIN
function - positions the file pointers at the beginning of a spin

SYNOPSIS
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT start_of_spin (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, SDDAS_SHORT ctrl_sen,
SDDAS_SHORT etime_yr, SDDAS_SHORT etime_day,
SDDAS_LONG etime_sec, SDDAS_LONG etime_nano)

ARGUMENTS
- data_key - unique value which indicates the data set of interest
- exten - two character extension to be added to IDFS file names when
  default files are not to be used, otherwise a null string
- version - IDFS data set identification number which allows for multiple
  openings of the same data set
- ctrl_sen - sensor which serves as the controller for timing relating to
  full spins
- etime_yr - ending year for data being requested
- etime_day - ending day of year for data being requested
- etime_sec - ending time of day in seconds for data being requested
- etime_nano - ending time of day residual in nanoseconds
- start_of_spin - routine status (see TABLE 1)

TABLE 1. Status Code Returned for START_OF_SPIN

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>START_SPIN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>START_SPIN_NO_POS</td>
<td>user did not call file_pos for this combination before calling this module</td>
</tr>
<tr>
<td>WRONG HEADER_FORMAT</td>
<td>multi-dimensional IDFS data storage is not supported by this module</td>
</tr>
<tr>
<td>START_SPIN_NO_SPIN</td>
<td>the requested data set does not spin</td>
</tr>
<tr>
<td>START_SPIN_MALLOC</td>
<td>no memory for structures which hold start of spin information for each sensor</td>
</tr>
<tr>
<td>START_SPIN_ALL_MALLOC</td>
<td>no memory for array of pointers to the idf_data structures that are allocated to hold the data for each sweep within the spin</td>
</tr>
<tr>
<td>START_SPIN_ETIME</td>
<td>user-requested end time was reached and the start of spin was not found</td>
</tr>
<tr>
<td>START_SPIN_TIME_MALLOC</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data being returned</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>Error codes returned by create_idf_data_structure ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by file_open ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by file_pos ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by read_drec ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by next_file_start_time ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by reset_experiment_info ()</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>
DESCRIPTION

Start_of_spin is the IDFS routine that positions the file descriptors at the start of a spin. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The start of spin can be determined in one of two ways: (1) use the azimuthal angle values (default method) or (2) use the spin periods defined by the IDFS data source that was specified as the start of spin source in the VIDF file for the data set of interest. If there was a problem encountered for the start of spin IDFS source on the initial call to the file_open module for the data set of interest, the start of spin method will be reverted back to the angular method. Both methods require that the start_of_spin module be called AFTER a call to the file_pos routine has been made since the file_pos routine sets the data pointer to the data sample or sweep whose beginning time is closest to that requested by the user, including any start of spin IDFS source that may be defined.

For the angular method, this routine uses the currently opened files for the requested data set and sets the current data pointer to the data sample or element within the sweep whose azimuthal angles indicate the start of a new spin has been reached (angle = 0.0). The start of spin is found for each individual IDFS sensor that is to be processed for the data set specified (refer to the select_sensor routine). This is necessary since the computation for the azimuthal angles is based upon time and each IDFS sensor could potentially start at a different time within the same sweep being retrieved from the IDFS data record.

For the start of spin source definition method, this routine uses the currently opened files for the requested data set and sets the current data pointer to the data sample or element within the sweep whose time period lies within the spin period that is closest to the user requested start time. The start of spin is found for each individual IDFS sensor that is to be processed for the data set specified (refer to the select_sensor routine). This is necessary since each IDFS sensor could potentially start at a different time within the same sweep being retrieved from the IDFS data record.

In preparation for the usage of spin-averaged data which is returned by the modules defined in section 2R, the parameter ctrl_sen is utilized. Spin-averaged data refers to data that is averaged over a complete spin. For the angular method, the parameter ctrl_sen defines the sensor that will dictate the time interval for each spin that is processed by the spin-averaging software. This is necessary since each IDFS sensor could potentially start at a different time within the same sweep being retrieved from the IDFS data record; therefore, one sensor has to control the time period that is reflective of each spin. For the start of spin source definition method, the parameter ctrl_sen is ignored; instead, the sensor that is defined in the StartOfSpin structure within the VIDF file is used. If the modules defined in section 2R are not going to be utilized, the user is advised to set this parameter to zero.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number
instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
file_pos 1R
read_drec 1R
select_sensor 1R
get_data_key 1R
get_version_number 1R
create_idf_data_structure 1R
next_file_start_time 1R
reset_experiment_info 1R
spin_data 2R
spin_data_pixel 2R
ret_codes 1H
libbase_idfs 1H

BUGS
None

EXAMPLES
Position the default IDFS data files associated with the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project, at the start of the spin that is closest to the user-requested start time specified in the prior call to file_open () and file_pos () modules.
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_LONG etime_sec, etime_nsec;
SDDAS_USHORT vnum;
SDDAS_SHORT status, etime_yr, etime_day;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

etime_yr = 1992;
etime_day = 217;
etime_sec = 32342;
etime_nsec = 0;
.
.
status = start_of_spin (data_key, ",", vnum, 0, etime_yr, etime_day, etime_sec, etime_nsec);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by start_of_spin routine.\n", status);
    exit (-1);
}
TURN_OFF_PITCH_ANGLE_COMPUTATIONS

function – disables the computation of pitch angles for the specified IDFS data set

SYNOPSIS

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT turn_off_pitch_angle_computations (SDDAS_ULONG data_key,
SDDAS_CHAR *exten, SDDAS_USHORT version)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple openings of the same data set

TABLE 1. Status Codes Returned for TURN_OFF_PITCH_ANGLE_COMPUTATIONS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TURN_OFF_PA_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Turn_off_pitch_angle_computations is the routine that disables the computation of the pitch angle data for the data set of interest. This routine was developed in order to speed up the read_drec routine since there are times when pitch angle data is not needed by the application accessing the IDFS data. The default scenario for the IDFS data access software is to compute and return pitch angle data, if available, for the data set of interest. If the data set of interest does not return pitch angle data, this routine has no effect. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. If this module is utilized, it must be called after the file_open routine has been called and before the file_pos routine is called.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable `USER_DATA`, which is set by the user, or in the user's home directory if the environment variable `USER_DATA` is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `file_pos` 1R
- `read_drec` 1R
- `get_data_key` 1R
- `get_version_number` 1R
- `ret_codes` 1H
- `libbase_idfs` 1H

**BUGS**

None

**EXAMPLES**

Turn off the pitch angle computations for the virtual instrument CP3DRH, which is part of the 3DR instrument, which is part of the PEACE experiment, which is part of the CLUSTER-2 mission, which is identified with the CLUSTERII project.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
```
status = get_data_key ("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = file_open (data_key, ",", vnum, -1, -1, -1, 0, -1, -1, -1, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_open routine.\n", status);
    exit (-1);
}

status = turn_off_pitch_angle_computations (data_key, ",", vnum);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by turn_off_pitch_angle_computations routine.\n", status);
    exit (-1);
}
TURN_ON_CELESTIAL_POSITION_COMPUTATIONS

function – enables the computation of celestial position angles for the specified IDFS data set

SYNOPSIS

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT turn_on_celestial_position_computations (SDDAS_ULONG data_key,
SDDAS_CHAR *exten, SDDAS_USHORT version)

ARGUMENTS

data_key - unique value which indicates the data set of interest

exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string

version - IDFS data set identification number which allows for multiple openings of the same data set

turn_on_celestial_position_computations - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for TURN_ON_CELESTIAL_POSITION_COMPUTATIONS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TURN_ON_CP_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Turn_on_celestial_position_computations is the routine that enables the computation and return of the celestial position angle data for the data set of interest. This routine was developed in order to control the ancillary data computations performed by the read_drec routine since there are few times when celestial position angle data is needed by the application accessing the IDFS data. The default scenario for the IDFS data access software is to suppress the computation of celestial position angle data, if available, for the data set of interest. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. If this module is utilized, it must be called after the file_open routine has been called and before the file_pos routine is called.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number.
instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
file_pos 1R
read_drec 1R
get_data_key 1R
get_version_number 1R
ret_codes 1H
libbase_idfs 1H

BUGS
None

EXAMPLES

Turn on the celestial position angle computations for the virtual instrument CP3DRH, which is part of the 3DR instrument, which is part of the PEACE experiment, which is part of the CLUSTER-2 mission, which is identified with the CLUSTERII project.

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_UINT data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
status = get_data_key ("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH",
&data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = file_open (data_key, "", vnum, -1, -1, 0, -1, -1, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_open routine.\n", status);
    exit (-1);
}

status = turn_on_celestial_position_computations (data_key, "", vnum);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by turn_on_celestial_position_computations routine.\n", status);
    exit (-1);
}
turn_on_celestial_position_computations (1R)  turn_celestial_position_computations (1R)
TURN_ON_EULER_ANGLE_COMPUTATIONS

function – enables the computation of euler angles for the specified IDFS data set

SYNOPSIS

```
#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT turn_on_euler_angle_computations (SDDAS_ULONG data_key,
                                              SDDAS_CHAR *exten, SDDAS_USHORT version)
```

ARGUMENTS

data_key  - unique value which indicates the data set of interest
exten     - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version   - IDFS data set identification number which allows for multiple openings of the same data set
turn_on_euler_angle_computations - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for TURN_ON_EULER_ANGLE_COMPUTATIONS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TURN_ON_EA_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

**Turn_on_euler_angle_computations** is the routine that enables the computation and return of the euler angle data for the data set of interest. This routine was developed in order to control the ancillary data computations performed by the read_drec routine since there are few times when euler angle data is needed by the application accessing the IDFS data. The default scenario for the IDFS data access software is to suppress the computation of euler angle data, if available, for the data set of interest. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. If this module is utilized, it must be called after the file_open routine has been called and before the file_pos routine is called.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
file_pos 1R
read_drec 1R
get_data_key 1R
get_version_number 1R
ret_codes 1H
libbase_idfs 1H

BUGS
None

EXAMPLES
Turn on the euler angle computations for the virtual instrument CP3DRH, which is part of the 3DR instrument, which is part of the PEACE experiment, which is part of the CLUSTER-II mission, which is identified with the CLUSTERII project.

#include "libbase_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
status = get_data_key ("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH", 
&data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = file_open (data_key, ",", vnum, -1, -1, 0, -1, -1, -1, 0, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by file_open routine.\n", status);
    exit (-1);
}

status = turn_on_euler_angle_computations (data_key, ",", vnum);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by turn_on_euler_angle_computations routine.\n", status);
    exit (-1);
}
valid_idf_data_structure (1R)

VALID_IDF_DATA_STRUCTURE
    function – indicates if the specified idf_data structure is an active data structure

SYNOPSIS
    #include "libbase_idfs.h"

    SDDAS_BOOL   valid_idf_data_structure (void *idf_data_ptr)

ARGUMENTS
    idf_data_ptr  - pointer to the idf_data structure that is to hold sensor
data and pertinent ancillary data for the data set of interest

DESCRIPTION
    Valid_idf_data_structure is the IDFS routine that can be called to verify that the address
of the data structure idf_data is a valid and active address. In other words, the memory
associated with the address has not been freed. While it is not usually necessary to utilize
this module when processing IDFS data, there are some cases where it was necessary when
SCF data is being processed. The module will return a Boolean value to indicate whether
the address for the idf_data structure is “active” at this time (sTrue) or “inactive” at this
time (sFalse).

ERRORS
    This routine returns no error codes.

SEE ALSO
    create_data_structure  1R
    create_idf_data_structure  1R
    idf_data  1S

BUGS
    None

EXAMPLES
    Determine if the memory address for the idf_data structure is an active memory location.
    If it is not, allocate a new idf_data structure for usage. This code segment assumes that the
module create_idf_data_structure or create_data_structure has been previously called.

    #include "libbase_idfs.h"

    SDDAS_SHORT status;
    SDDAS_BOOL valid_address;
    void *idf_data_ptr;

    valid_address = valid_idf_data_structure (idf_data_ptr);
if (!valid_address)
{
    status = create_idf_data_structure (&idf_data_ptr);
    if (status != ALL_OKAY)
    {
        printf("\n Error %d returned by create_idf_data_structure routine.\n", status);
        exit (-1);
    }
}
VALID_TENSOR_DATA_STRUCTURE
function – indicates if the specified tensor_data structure is an active data structure

SYNOPSIS
#include "libbase_idfs.h"

SDDAS_BOOL valid_tensor_data_structure (void *tensor_data_ptr)

ARGUMENTS
tensor_data_ptr - pointer to the tensor_data structure that is to hold sensor data and pertinent ancillary data for the data set of interest

DESCRIPTION
Valid_tensor_data_structure is the IDFS routine that can be called to verify that the address of the data structure tensor_data is a valid and active address. In other words, the memory associated with the address has not been freed. While it is not usually necessary to utilize this module when processing IDFS data, there are some cases where it was necessary when SCF data is being processed. The module will return a Boolean value to indicate whether the address for the tensor_data structure is “active” at this time (sTrue) or “inactive” at this time (sFalse).

ERRORS
This routine returns no error codes.

SEE ALSO
create_data_structure 1R
create_tensor_data_structure 1R
tensor_data 1S

BUGS
None

EXAMPLES
Determine if the memory address for the tensor_data structure is an active memory location. If it is not, allocate a new tensor_data structure for usage. This code segment assumes that the module create_tensor_data_structure or create_data_structure has been previously called.

#include "libbase_idfs.h"

SDDAS_SHORT status;
SDDAS_BOOL valid_address;
void *tensor_data_ptr;

valid_address = valid_tensor_data_structure (tensor_data_ptr);
if (!valid_address)
{
    status = create_tensor_data_structure (&tensor_data_ptr);
    if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by create_tensor_data_structure routine.\n", status);
        exit (-1);
    }
}
BUFFER_BIN_FILL

function - fills in the missing bin elements for the data buffer being referenced

SYNOPSIS

#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT buffer_bin_fill (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, SDDAS_FLOAT *data_ptr,
SDDAS_CHAR *bin_stat, SDDAS_SHORT block_size,
SDDAS_LONG need_filled, SDDAS_CHAR bin_project)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten   - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version   - IDFS data set identification number which allows for multiple
openings of the same data set
data_ptr  - pointer to the data buffer being processed
bin_stat  - pointer to status flags which are associated with each data
bin returned
block_size  - the number of data values returned in a single data buffer
need_filled  - the number of filled bins needed in order to fill in the missing
data bins
bin_project  - flag indicating if the data is to be projected into empty bins
between the first or last data bin which contains data
buffer_bin_fill - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for BUFFER_BIN_FILL

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUF_BIN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for</td>
</tr>
<tr>
<td></td>
<td>processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>BUF_BIN_MALLOC</td>
<td>no memory for temporary internal array</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Buffer_bin_fill is the IDFS routine which fills any missing bins in the data buffers that are
returned by the IDFS routines that return time-averaged data (fill_data and
fill_discontinuous_data), sample-averaged data (sweep_data and
sweep_discontinuous_data) or spin-averaged data (spin_data and spin_data_pixel). The user should process only those buffers that are flagged with the status value BUFFER_READY. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. This module can not be used in conjunction with the fill_mode_data and sweep_mode_data modules since this module is concerned with sensor-specific data.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

The empty data bins will be filled in according to the method selected in the call to the set_bin_info module, with the exception of a least squares fit. The least squares fit method, if selected, is replaced by a constant column row approach, with a minimum of 3 data points needed before the fill in can be accomplished. This replacement is necessary since the least squares fit method is valid for 2-D fits only and this routine works with 1-D data. If the user selected NO_BIN_FILL, the data bins are left as is, with any unfilled bins left unfilled. If the user selected any other fill method, that fill method is used to fill in the missing data bins. If the user is collapsing the data over any data dimension, there is no need to call this module. The missing data bins will be handled by the call to the collapse_dimensions module.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.
SEE ALSO

file_open               1R
fill_data               2R
fill_discontinuous_data 2R
sweep_data              2R
sweep_discontinuous_data 2R
spin_data               2R
spin_data_pixel         2R
collapse_dimensions     2R
set_bin_info            2R
get_data_key            1R
get_version_number      1R
ret_codes               1H
libtrec_idfs             2H

BUGS

None

EXAMPLES

Assume that the fill_data module has been called and that data_ptr has been set to point to a data buffer that has been flagged as BUFFER_READY and bin_stat has been set to point to the corresponding status flags for the data buffer in question. The variable data_block is returned by the module fill_data. Use a minimum of three data points to fill in the missing bins and do not project the data past the first and last data bins actually found. The data set selected is from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status, data_block;
SDDAS_FLOAT *data_ptr;
SDDAS_CHAR *bin_stat;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\nError %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = buffer_bin_fill (data_key, "", vnum, data_ptr, bin_stat, data_block, 3, 0);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by buffer_bin_fill routine.\n", status);
    exit (-1);
}
CENTER_AND_BAND_VALUES
function - creates the center sweep and/or band width values associated with the data bins

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT  center_and_band_values (SDDAS_ULONG data_key,
        SDDAS_CHAR *exten, SDDAS_USHORT version,
        void *idf_data_ptr, SDDAS_SHORT sensor,
        SDDAS_CHAR ret_center, SDDAS_CHAR ret_band,
        SDDAS_FLOAT **center_ptr, SDDAS_FLOAT **low_ptr,
        SDDAS_FLOAT **high_ptr, SDDAS_SHORT *num_bins,
        SDDAS_SHORT *num_converted)

ARGUMENTS
data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple openings of the same data set
idf_data_ptr - pointer to the idf_data structure that is to hold sensor data and pertinent ancillary data for the data set of interest
sensor - sensor identification number
ret_center - flag indicating if the center sweep values are to be returned
    0 - do not return the center sweep value
    1 - calculate and return the pointer to the center sweep values
ret_band - flag indicating if the band width values for the sweep are to be returned
    0 - do not return the band width values for the sweep
    1 - calculate and return the pointer(s) to the band width values for the sweep
center_ptr - pointer to the location that holds the center sweep values
low_ptr - pointer to the location that holds the lower bands for non-contiguous bands or all band widths for contiguous bands
high_ptr - pointer to the location that holds the upper bands for non-contiguous bands
num_bins - the number of values returned

197  December 28, 2012
Center_and_band_values is the IDFS routine that creates the center and/or band width sweep step values for the specified sensor. For any given virtual instrument, there may be one set of sweep step values to be used by all sensors or there may be a set of sweep step values defined for each individual sensor. In either case, this routine should be called once for each sensor that is to be processed by the IDFS routines that return time-averaged data (fill_data / fill_discontinuous_data), sample-averaged data (sweep_data / sweep_discontinuous_data) or spin-averaged data (spin_data and spin_data_pixel). These sweep step values are used by the time-averaging, sample-averaging or spin-averaging module when storing the data into the data bins for VARIABLE_SWEEP processing (refer to the explanation in the set_bin_info write-up), but are not used for FIXED_SWEEP processing. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. If the only type of data to be processed for the data set in question is instrument status (mode) data, the user does not need to call this module since the center_and_band_values routine processes sensor-specific data and instrument status data is not sensor-specific.

The sweep step values are created using the information specified by the calls to the set_bin_info and set_scan_info modules. If the set_bin_info module has not been called, an error code is returned to the calling module. If the set_scan_info module has not been called, the center and band width values will be calculated in terms of raw units and the IDFS software will set up the system so that only one set of sweep step values are defined for all sensors for the data set selected. When FIXED_SWEEP processing is specified in the call to the set_bin_info module, there may be cases when all the center and band values are not converted to the units desired. This situation can occur if the unit conversion is dependent upon calibration data that is stored within the data record. In this case, the
number of center and band values that are converted to units is dependent upon the number of data values contained within the first data record processed. This number is returned in the \texttt{num\_converted} parameter and is strictly for informational purposes only and this module returns the status code \texttt{CENTER\_CONVERSION}.

The parameter \texttt{version} allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the \texttt{get\_version\_number} routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter \texttt{idf\_data\_ptr} is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the \texttt{create\_idf\_data\_structure} routine is made. The user also has the option of calling the module \texttt{create\_data\_structure}, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the \texttt{create\_idf\_data\_structure} routine \texttt{N} times to create \texttt{N} instances of the \texttt{idf\_data} structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the \texttt{file\_open} routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable \texttt{exten}. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable \texttt{USER\_DATA}, which is set by the user, or in the user's home directory if the environment variable \texttt{USER\_DATA} is not set. To open the default IDFS data files, \texttt{exten} should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set \texttt{exten} to a null string for real-time scenarios.

The contents of the memory locations returned by this module should \textbf{NOT} be altered since the calculated center/band width values are used by the time-averaging, sample-averaging, or spin-averaging routine when processing the data. If the returned values need to be modified, for example, to take the log of the values, the user should allocate space to hold the values, copy the values into this space and modify the values there.

The module returns two possible pointers for the location(s) that hold the lower and upper band width values. In the case where the bands are non-contiguous, both the \texttt{low\_ptr} and \texttt{high\_ptr} will reference memory locations that hold the band width values. In the case where the bands are contiguous, there is no need to hold separate upper and lower values –
the upper limit of the current band is the lower limit of the next band. In this case, one extra memory location is allocated, the `high_ptr` pointer is set to nil or 0 and `low_ptr` is set to reference the location that holds the band width values.

An instrument may be classified as a vector instrument or as a scalar instrument. A vector instrument is an instrument whose sensors represent multi-value data sets as opposed to a scalar instrument whose sensors represent a set of singular data values. If the data set to be processed is a vector instrument, the user should not call the routine `select_sensor` for this data set; otherwise, erroneous center sweep and/or band width values may be computed.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `convert_to_units` 1R
- `fill_data` 2R
- `fill_discontinuous_data` 2R
- `sweep_data` 2R
- `sweep_discontinuous_data` 2R
- `spin_data` 2R
- `spin_data_pixel` 2R
- `get_data_key` 1R
- `set_bin_info` 2R
- `set_scan_info` 2R
- `get_version_number` 1R
- `select_sensor` 1R
- `create_data_structure` 1R
- `create_idf_data_structure` 1R
- `ret_codes` 1H
- `libtrec_idfs` 2H

**BUGS**

None

**EXAMPLES**

Create the center and band width sweep values for sensor zero for the RTLA virtual instrument, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libbase_idfs.h"
#include "ret_codes.h"
```
SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT sensor, status, num_bins, num_converted, bin;
SDDAS_FLOAT *center_ptr, *low_ptr, *high_ptr;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

sensor = 0;
status = center_and_band_values (data_key, 
"", vnum, idf_data_ptr, sensor, 1, 1, 
&center_ptr, &low_ptr, &high_ptr, &num_bins, 
&num_converted);
if (status != ALL_OKAY && status != CENTER_CONVERSION)
{
    printf ("\n Error returned by center_and_band_values.\n");
    exit (-1);
}

/* Bands are contiguous? */
if (high_ptr == NULL)
    for (bin = 0; bin < num_bins; ++bin)
        printf ("\nlow = %f high = %f", *(low_ptr + bin), *(low_ptr + bin + 1));

/* Bands are non-contiguous. */
else
    for (bin = 0; bin < num_bins; ++bin)
        printf ("\nlow = %f high = %f", *(low_ptr + bin), *(high_ptr + bin));
**COLLAPSE_DIMENSIONS**

function - collapses data over the requested dimensions for a single data level (unit)

**SYNOPSIS**

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT collapse_dimensions (SDDAS_ULONG data_key,
                                SDDAS_CHAR *exten, SDDAS_USHORT version,
                                SDDAS_SHORT sensor, SDDAS_CHAR *dimen,
                                SDDAS_FLOAT *s_range, SDDAS_FLOAT *e_range,
                                SDDAS_CHAR avg_type, SDDAS_CHAR int_type,
                                SDDAS_FLOAT **ret_data, SDDAS_CHAR cyclic,
                                SDDAS_SHORT order, SDDAS_LONG need_filled,
                                SDDAS_FLOAT tension, SDDAS_CHAR norm_res,
                                SDDAS_CHAR bin_project, SDDAS_SHORT unit_index,
                                SDDAS_CHAR last_plot, SDDAS_CHAR dlevel,
                                SDDAS_CHAR cur_buf)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **sensor** - sensor identification number
- **dimen** - status flag indicator for each possible dimension
- **s_range** - the starting range value for each possible dimension
- **e_range** - the ending range value for each possible dimension
- **avg_type** - the scheme to use in order to reduce the dimensionality of the data

1 - no reduction is to be performed (NO_AVG)
2 - a straight average is to be performed (STRAIGHT_AVG)
3 - a straight integration is to be performed (STRAIGHT_INT)
4 - a spherical integration is to be performed (SPHERICAL_INT)
5 - a straight average is to be performed assuming the data represents azimuthal angle values (STRAIGHT_AVG_AZ)
6 - a flux integration is to be performed (FLUX_INT)
7 - moments computation is to be performed (MOMENTS_INT)
int_type - integration scheme to use for calculations
  1 - a trapezoidal integration (POINT_INT)
  2 - a block integration (BAND_INT)
ret_data - pointer to the resultant matrix or value
cyclic - flag indicating if the data is cyclic
  0 - data is not cyclic
  1 - data is cyclic
order - the order of the fit, i.e. 1, 2, 3, etc.
  - this parameter is used if the bin fill method
    chosen in the call to the set_bin_info
    routine is any value other than NO_BIN_FILL.
need_filled - the number of filled bins needed in order to fill
  in the missing data bins
tension - the weighting of the data
norm_res - flag indicating if the result is to be normalized
  0 - do not normalize the result
  1 - normalize the result
bin_project - flag indicating if the data is to be projected
  into empty bins beyond the first or last data bin which contains data
  0 - do not project the data
  1 - project the data
unit_index - index value specifying which sub-buffer returned from
  the fill_data / fill_discontinuous_data / sweep_data /
  sweep_discontinuous_data / spin_data / spin_data_pixel
  routine is to be processed
last_plot - flag indicating if this call to collapse_dimensions is
  the last call to be made for the combination being processed
  (necessary for reset purposes)
  0 - not the last call for the combination being processed
  1 - the last call for the combination being processed
dlevel - flag indicating if data is to be reduced to a single value
  or to an array of values.
  1 - data to be reduced to a single value.
  2 - data to be reduced to an array of values (sweep of values)
cur_buf - the current buffer being processed (number between 0 and
  NUM_BUFFERS-1)
collapse_dimensions - routine status (see TABLE 1)
### TABLE 1. Status Codes Returned for COLLAPSE DIMENSIONS

<table>
<thead>
<tr>
<th>STATUS_CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDIMEN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>COMPUTE_MOMENTS</td>
<td>the routine moments_computations() must be called when requesting the moments integration averaging method</td>
</tr>
<tr>
<td>CDIMEN_COLLAPSE</td>
<td>no memory has been allocated to hold the collapsing information (user did not call setCollapse_info for this combination)</td>
</tr>
<tr>
<td>CDIMEN_MANY_SCAN</td>
<td>the requested data set has more than one scan range defined</td>
</tr>
<tr>
<td>CHRG_PA_ERROR</td>
<td>the Straight Average Azimuthal averaging option for the Charge dimension is not a valid averaging method</td>
</tr>
<tr>
<td>MASS_PA_ERROR</td>
<td>the Straight Average Azimuthal averaging option for the Mass dimension is not a valid averaging method</td>
</tr>
<tr>
<td>NEG_BIN_STAT</td>
<td>the base buffer has a bin status value that is negative</td>
</tr>
<tr>
<td>TRANS_3D_BINNED_MALLOC</td>
<td>no memory has been allocated to hold the normalization factors needed to combine those sensors that are mounted at the same theta angles</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

### DESCRIPTION

**Collapse_dimensions** is the IDFS routine that reduces the dimensionality of the data returned by the time-averaging routine (fill_data / fill_discontinuous_data), the sample-averaging routine (sweep_data / sweep_discontinuous_data) or the spin-averaging routine (spin_data / spin_data_pixel). The user should process only those buffers that are flagged with the status value BUFFER_READY. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. This module cannot be used in conjunction with the fill_mode_data / sweep_mode_data module since dimensionality is associated with sensor-specific data and instrument status data is not sensor-specific.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null
string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

This routine will work on one data level or "unit". If there are many data levels to be processed, there should be multiple calls to this routine, all made after the time-averaging, sample-averaging, or spin-averaging module has been called. The address of the space that holds the result is passed back to the user. This may be a single value or may be an array of values, as indicated by the `dlevel` parameter. The value for the parameter `unit_index` can be retrieved by calling the module `units_index`, with the third argument from the end holding the value to be passed to this module. If no data is present in the data buffer being processed, the module will return the data value(s) -3.4e38 (OUTSIDE_MIN).

The dimensions that are to be collapsed over are specified in the array `dimen`. The `s_range` and `e_range` parameters are arrays that hold the starting and ending ranges to use to reduce the data at each possible dimension. All three arrays are order dependent, with the order as follows:

- element 0 range for the scan dimension
- element 1 range for the theta dimension
- element 2 range for the phi dimension
- element 3 range for the mass dimension
- element 4 range for the charge dimension
- element 5 range for scalar averaging (sensors to be averaged together)

If a given dimension is to be collapsed over, the value within the `dimen` array corresponding to that dimension should be set to one. If the dimension is not be collapsed over, with no impact on the result, the value within the `dimen` array corresponding to the dimension should be set to zero. If the hemisphere assumption factor is to be utilized for a given dimension, the value within the `dimen` array corresponding to that dimension should be set to two. The hemisphere assumption factor is dependent upon the dimension being processed and the scheme (method) selected in order to reduce the dimensionality of the data. The hemisphere assumption factor is not selectable by the user. The hemisphere assumption factors were obtained for each method assuming angular isotropy. The table below summarizes the values used for assumption factors during processing:

### TABLE 2. Hemisphere Assumption Factors

<table>
<thead>
<tr>
<th>DIMENSION</th>
<th>STRAIGHT INTEGRATION</th>
<th>SPHERICAL INTEGRATION</th>
<th>FLUX INTEGRATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARGE</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>MASS</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>PHI</td>
<td>n/a</td>
<td>2\pi</td>
<td>2\pi</td>
</tr>
<tr>
<td>THETA</td>
<td>n/a</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>SCAN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

The scalar average selection should be set when an average of the sensor data values for scalar instruments is desired. If the virtual instrument is designed such that there is more...
than one scan range defined for all sensors, the error code **CDIMEN_MANY_SCAN** will be returned if any dimension other than the scan dimension is selected.

Before the actual collapsing is performed, the missing bins are filled in according to the method specified with the call to the module `set_bin_info`. The actual reduction of the data can be performed in one of three ways: straight average, straight integration and spherical integration. The straight average is the simplest of the three schemes - a simple average of the data between the start and stop range specified. A special straight average algorithm is provided for azimuthal angle data. This algorithm takes into account the roll over to a minimum value (0º) when the maximum threshold (360º) has been reached or to a maximum value when the minimum threshold has been reached. If a straight average is not appropriate, the data can be reduced by integrating over the range specified. The distinction between a straight and spherical integration is the integration over the sensors. The spherical integration method may be appropriate when the sensors represent discrete theta angular ranges. With the integration reduction, the user may also select the integration scheme, either point or band integration. A point integration is a trapezoidal integration using the center of each bin as the integration parameter. For band integration, the bin widths of each bin are used as the integration widths in a rectangular or block integration.

Since missing bins are filled in prior to data reduction, the data matrices must be reset to their original contents prior to the next call to the time-averaging, sample-averaging, or spin-averaging routine in order to utilize the interleave option. In order for the module to know when the data is to be reset, the module examines the contents of the `last_plot` parameter. When the contents is set to a one, the data matrices are restored. This parameter should be set once the last data level is being processed for the data set being requested; otherwise, erroneous calculations will result.

The parameter `avg_type` specifies the method that is to be used to reduce the dimensionality of the data. The range of possible values for the `avg_type` parameter has been expanded since the ability to compute moments has been added to the IDFS data access capabilities. In order to compute the moments value(s), the dimensionality of the data may need to be reduced. However, the data reduction can not be performed by the `collapse_dimensions` routine and is trapped as an error if this function is called. At the present time, the IDFS Programmers Manual does not contain any information regarding the moments computations; however, this information will be added once testing of the moments computation software has been completed.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `fill_data` 2R
EXAMPLES

Collapse the data returned for sensor 1 for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. The data is collapsed over the frequency range 0.16 to 0.9 kilohertz using a straight average. Assume that only one data level or unit is returned by the time-averaging, sample-averaging, or spin-averaging routine (default mode) and that buf_stat had been set.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"
define DUMMY_VAL 0

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
SDDAS_FLOAT *data_ptr, start[6], stop[6];
SDDAS_CHAR dimen[6], cur_buf, buf_num, *buf_stat;
static SDDAS_CHAR which_buf = 0;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf("n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

dimen[0] = 1;
```
cur_buf = which_buf;
for (buf_num = 0; buf_num < NUM_BUFFERS; ++buf_num)
{
    if (*(buf_stat + cur_buf) == BUFFER_READY)
    {
        status = collapse_dimensions (data_key, "", vnum, 1, dimen, start, stop,
                                      STRAIGHT_AVG, DUMMY_VAL, &data_ptr, 0, 1, 3,
                                      0.0, 1, 1, 0, 1, 1, cur_buf);
        if (status != ALL_OKAY)
        {
            printf ("\n Error %d returned by collapse_dimensions routine.\n", status);
            exit (-1);
        }
        which_buf = (cur_buf + 1) % NUM_BUFFERS;
    }
    cur_buf = (cur_buf + 1) % NUM_BUFFERS;
}
FILL_DATA
function - returns time-averaged data buffers for data sets that do not roll over when the minimum/maximum threshold has been reached

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT fill_data (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, void *idf_data_ptr,
SDDAS_SHORT **ret_sensors, SDDAS_FLOAT **ret_data,
SDDAS_FLOAT **ret_frac, SDDAS_CHAR **bin_stat,
SDDAS_LONG **bpix, SDDAS_LONG **epix,
SDDAS_CHAR **ret_stat, SDDAS_SHORT *num_sen,
SDDAS_SHORT **num_units, SDDAS_SHORT *block_size,
SDDAS_SHORT **stime_yr, SDDAS_SHORT **stime_day,
SDDAS_LONG **stime_sec, SDDAS_LONG **stime_nano,
SDDAS_SHORT **etime_yr, SDDAS_SHORT **etime_day,
SDDAS_LONG **etime_sec, SDDAS_LONG **etime_nano,
SDDAS_CHAR *hdr_change,
SDDAS_UCHAR exclude_dqual)

ARGUMENTS
data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple openings of the same data set
idf_data_ptr - pointer to the idf_data structure that temporarily holds sensor data and pertinent ancillary data for the data set of interest
ret_sensors - an array which holds the sensor number(s) for which data is returned
ret_data - pointer to the data being returned (data for all sensors processed)
ret_frac - pointer to the normalization factors for the data being returned
bin_stat - pointer to status flags which are associated with each data bin returned
0 - no data has been placed into the data bin being processed
1 - data has been placed into the data bin being processed
bpix - pointer to the starting pixel location for the data buffers returned
epix - pointer to the ending pixel location for the data buffers returned
ret_stat - pointer to the status of each of the data buffers being returned
  UNTouched_BUFFER - no data has ever been placed into the buffer
  FREE_BUFFER - no data has been placed into the buffer being processed (ready for re-use)
  PARTIAL_WORKING - data is being acquired into the buffer but is not ready for processing
  BUFFER_READY - data has been acquired into the buffer and is ready for processing
num_sen - the number of elements in the ret_sensors array
num_units - an array holding the number of data sets to bypass in order to get to the data for the sensor being processed
block_size - the number of data values returned in a data buffer
stime_yr - pointer to the start time year values for the data buffers returned
stime_day - pointer to the start time day of year values for the data buffers returned
stime_sec - pointer to the start time of day values (in seconds) for the data buffers returned
stime_nano - pointer to the start time of day residuals (in nanoseconds) for the data buffers returned
etime_yr - pointer to the end time year values for the data buffers returned
etime_day - pointer to the end time day of year values for the data buffers returned
etime_sec - pointer to the end time of day values (in seconds) for the data buffers returned
etime_nano - pointer to the end time of day residuals (in nanoseconds) for the data buffers returned
hdr_change - flag which indicates a header change occurred while processing the data
  0 - a header change was not encountered during the processing of the data
  1 - a header change was encountered during the processing of the data
exclude_dqual - data is to be excluded if the d_qual flag associated with the data is set to the value specified
fill_data - routine status (see TABLE 1)
TABLE 1. Status Codes Returned for FILL_DATA

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for</td>
</tr>
<tr>
<td></td>
<td>processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which hold information pertinent to the time-averaged data</td>
</tr>
<tr>
<td>FILL_BASE_TIME_MISSING</td>
<td>the time interval information has not been set (user did not call set_time_values</td>
</tr>
<tr>
<td></td>
<td>for this combination)</td>
</tr>
<tr>
<td>FILL_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call set_bin_info</td>
</tr>
<tr>
<td></td>
<td>for this combination)</td>
</tr>
<tr>
<td>FILL_CENTER_BAND_MISSING</td>
<td>the routine center_and_band_values has not been called prior to calling the FILL_DATA</td>
</tr>
<tr>
<td></td>
<td>routine</td>
</tr>
<tr>
<td>FILL_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>FILL_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_SWP_MALLOC</td>
<td>no memory for FILL_SWP_REALLOC</td>
</tr>
<tr>
<td>FILL_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>FILL_DATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>FILL_WITH_SWEEP</td>
<td>the modules fill_data and sweep_data cannot be used interchangeably for the same data</td>
</tr>
<tr>
<td></td>
<td>key, extension, version combination</td>
</tr>
<tr>
<td>BAD_VFMT</td>
<td>bad format character for variable width bin spacing</td>
</tr>
<tr>
<td>NO_EMPTY_BUFFERS</td>
<td>no spare buffers for data accumulation</td>
</tr>
<tr>
<td>PHI_DIFF_UNITS</td>
<td>the sensors being processed do not process the same number of data levels (units)</td>
</tr>
<tr>
<td>FILL_PHI_FIRST</td>
<td>the starting azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td>FILL_PHI_LAST</td>
<td>the ending azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by read_drec ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by convert_to_units ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by fill_sensor_info ()</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

**Fill_data** is the IDFS time-averaging data read routine, retrieving data for all sensors that return data for the time duration being processed. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. **Fill_data** processes sensor-specific data only, that is, it processes sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. If the instrument status (mode) data is desired, the user should use the fill_mode_data routine. **Fill_data** assumes that the data set of interest does not roll over to a minimum value when the maximum threshold has been reached or to a maximum value when the minimum threshold has been reached. This assumption is crucial since multiple samples may be averaged together in a single buffer. If the data set does roll over at the thresholds, the averaging of these samples will probably result in incorrect data values. An example of a roll over data set is longitude data, which resets values to the minimum threshold (-180) when the maximum threshold (180) has been reached. If the data set does roll over, the user should use the fill_discontinuous_data routine. If the data set of interest is a combination of roll over and non-roll over data, for example, longitude data being returned along with science data, the user may use the fill_discontinuous_data module in
conjunction with the fill_data routine, using the fill_data routine to return the non-roll over data values and using the fill_discontinuous_data routine to return the roll over data values. In order to do this correctly, the user must make use of multiple version numbers so that the same data files can be opened more than once. That is, use one version number for the non-roll over data and another version number for the roll over data. All IDFS routines that utilize a version number must be called once for each unique version number.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter idf_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the create_idf_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_idf_data_structure routine N times to create N instances of the idf_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

This routine will process data one sweep at a time, placing the data into buffers which hold data that is accumulated over a specified time interval. Once the time interval has been processed, the routine will return the data buffers and a status value for each buffer which indicates when the buffer is ready for the user to retrieve. The user must call the module set_time_values before the fill_data module can be called since the set_time_values module is used to specify the base time value and reference location and the time interval (delta) to use to accumulate the data. If the fill_data routine determines that the set_time_values module has not been called, an error code is returned to the user.
Along with the data being returned, there is a starting location and an ending location that is returned for each of the data buffers. The user may use these values as references to the base location specified in the call to the `set_time_values` module. That is, given a base time value, a time interval and a reference location, the `fill_data` routine will return the location of data with respect to time. The user may choose to ignore these values or may use these locations to plot data along an axis that is scaled with respect to time.

There are a constant number of data buffers that are used by the `fill_data` module. This number is defined as `NUM_BUFFERS` in the `user_defs.h` file. This file is described in section 1H of the IDFS Programmers Manual. These data buffers are utilized in a cyclic nature, with buffer 0 being re-used once buffer `NUM_BUFFERS-1` has been filled. The data buffers that are ready to be processed are flagged with the status value `BUFFER_READY`. For each buffer, there are `N` many sub-buffers which hold the data in each of the requested data levels or units. The user must process the data contained within these buffers before the next call to the `fill_data` module is made since the module will clear out these buffers for re-use. This holds true even when an `LOS_STATUS` or `NEXT_FILE_STATUS` status code is returned. The data values must be normalized using the normalization factors returned along with the data. Since the buffers are cyclic, the user may wish to keep a variable indicating the last buffer number processed so that the user can start at the time sample left off from the previous call to the `fill_data` module at the next call. It is important to note that there is one status flag per data buffer that is used by all sensors. If the sensors rotate or alternate when data is returned, the result may be that a buffer is flagged as `BUFFER_READY` but may not contain any data since the data buffers are reset or cleared out upon each call to the `fill_data` module. The user is advised to check the value or values in the `bin_stat` array. If all values are 0, no data was placed into the buffers.

The size and spacing of the data buffers are either defined by the user or by elements contained within the virtual instrument definition document. The user must call the `set_bin_info` module before calling the `fill_data` routine in order to specify how the binning of the data is to occur. In addition, the user must call the `center_and_band_values` module before calling the `fill_data` module. If the `fill_data` routine determines that no binning scheme has been selected, an error code is returned to the user.

The user should be aware that the data buffers that come back from the `fill_data` module are NOT modified as far as missing bins is concerned. If the user wishes to fill in the missing bins according to the method specified in the call to the `set_bin_info` module, the user must call the `buffer_bin_fill` module. If the data are collapsed over specified dimensions, the `buffer_bin_fill` module need not be called.

The default mode for the `fill_data` routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the `fill_sensor_info` module prior to calling the `fill_data` module. The user should make one call to the
**fill_sensor_info** module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

If the virtual instrument acquires data over the PHI dimension and the user wishes to average the data over a specified phi range, the **fill_data** routine must be used to acquire the phi data matrix. The user must call the module **set_collapse_info** prior to calling the **fill_data** module in order to specify the resolution of the phi bins and to specify if the interleave option is to be utilized when building the phi matrix.

The parameter **exclude_dqual** holds a single value that is compared against the d_qual value found in the header record for the sensor being processed. If the user wishes to exclude data that is flagged with a specific d_qual value, the user should set the **exclude_dqual** value to this specific value. If the user wishes to include all data encountered, the user should set the **exclude_dqual** value to 255.

**ERRORS**

All errors within this routine are returned through the status variable. The include file **ret_codes.h**, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The **ret_codes.h** file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- **file_open** 1R
- **read_drec** 1R
- **convert_to_units** 1R
- **set_time_values** 2R
- **fill_discontinuous_data** 2R
- **fill_mode_data** 2R
- **set_bin_info** 2R
- **center_and_band_values** 2R
- **fill_sensor_info** 2R
- **set_collapse_info** 2R
- **buffer_bin_fill** 2R
- **get_data_key** 1R
- **get_version_number** 1R
- **create_data_structure** 1R
- **create_idf_data_structure** 1R
- **ret_codes** 1H
- **user_defss** 1H
- **libtree_idfs** 2H

**BUGS**

None
EXAMPLES

Obtain time-averaged data from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
SDDAS_LONG *bpix, *epix;
SDDAS_SHORT *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change, *buf_stat;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by get_data_key routine.\n", status);
        exit (-1);
    }

get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
        exit (-1);
    }

status = fill_data (data_key, ",", vnum, idf_data_ptr, &sen_numbers, &ret_data,
                  &ret_frac, &ret_bin, &bpix, &epix, &buf_stat, &num_sen,
                  &num_units, &data_block, &start_time_yr, &start_time_day,
                  &start_time_sec, &start_time_nano, &end_time_yr, &end_time_day,
                  &end_time_sec, &end_time_nano, &hdr_change, 255);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by fill_data routine.\n", status);
        exit (-1);
    }
```

217  December 28, 2012
**FILL_DATA_ENVELOPE**

function - returns the data envelope (minimum and maximum values) for the time interval being processed

**SYNOPSIS**

```
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT fill_data_envelope (SDDAS_ULONG data_key, SDDAS_CHAR *exten, 
   SDDAS_SHORT version, void *idf_data_ptr, 
   SDDAS_SHORT **ret_sensors, SDDAS_FLOAT **ret_min, 
   SDDAS_FLOAT **ret_max, SDDAS_CHAR **bin_stat, 
   SDDAS_LONG **bpix, SDDAS_LONG **epix, 
   SDDAS_CHAR **buf_stat, SDDAS_SHORT *num_sen, 
   SDDAS_SHORT **num_units, SDDAS_SHORT *block_size, 
   SDDAS_SHORT **stime_yr, SDDAS_SHORT **stime_day, 
   SDDAS_LONG **stime_sec, SDDAS_LONG **etime_nano, 
   SDDAS_SHORT **etime_yr, SDDAS_SHORT **etime_day, 
   SDDAS_LONG **etime_sec, SDDAS_LONG **etime_nano, 
   SDDAS_CHAR *hdr_change, 
   SDDAS_UCHAR exclude_dqual)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **idf_data_ptr** - pointer to the **idf_data** structure that temporarily holds sensor data and pertinent ancillary data for the data set of interest
- **ret_sensors** - an array which holds the sensor number(s) for which data is returned
  - the array is initialized to -1 in all elements; valid sensor numbers start with 0
- **ret_min** - pointer to the minimum data value encountered for the buffer(s) returned
- **ret_max** - pointer to the maximum data value encountered for the buffer(s) returned
- **bin_stat** - pointer to status flags which are associated with each data bin returned
  - 0 - no data has been placed into the data bin being processed
  - 1 - data has been placed into the data bin being processed
bpix - pointer to the starting pixel location for the data buffers returned
epix - pointer to the ending pixel location for the data buffers returned
buf_stat - pointer to the status of each of the data buffers being returned
    UNTOUCHED_BUFFER - no data has ever been placed into the buffer
    FREEBUFFER - no data has been placed into the buffer being processed (ready for re-use)
    PARTIAL_WORKING - data is being acquired into the buffer but is not ready for processing
    BUFFER_READY - data has been acquired into the buffer and is ready for processing
num_sen - the number of elements in the ret_sensors array
num_units - an array holding the number of data sets to bypass in order to get to the data for the sensor being processed
block_size - the number of data values returned in a data buffer
stime_yr - pointer to the start time year values for the data buffers returned
stime_day - pointer to the start time day of year values for the data buffers returned
stime_sec - pointer to the start time of day values (in seconds) for the data buffers returned
stime_nano - pointer to the start time of day residuals (in nanoseconds) for the data buffers returned
etime_yr - pointer to the end time year values for the data buffers returned
etime_day - pointer to the end time day of year values for the data buffers returned
etime_sec - pointer to the end time of day values (in seconds) for the data buffers returned
etime_nano - pointer to the end time of day residuals (in nanoseconds) for the data buffers returned
hdr_change - flag which indicates a header change occurred while processing the data
    0 - a header change was not encountered during the processing of the data
    1 - a header change was encountered during the processing of the data
exclude_dqual - data is to be excluded if the d_qual flag associated with the data is set to the value specified
fill_data_envelope - routine status (see TABLE 1)
TABLE 1. Status Codes Returned for **FILL_DATA_ENVELOPE**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_ENV_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call <code>file_open</code> for this combination)</td>
</tr>
<tr>
<td>FILL_ENV_SCALAR</td>
<td>the requested data source is non-scalar</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which hold information pertinent to the time-rectified data</td>
</tr>
<tr>
<td>FILL_ENV_BASE_TIME_MISSING</td>
<td>the time interval information has not been set (user did not call <code>set_time_values</code> for this combination)</td>
</tr>
<tr>
<td>FILL_ENV_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call <code>set_bin_info</code> for this combination)</td>
</tr>
<tr>
<td>FILL_ENV_CENTER_BAND_MISSING</td>
<td>the routine <code>center_and_band_values</code> has not been called prior to calling the <code>fill_data_envelope</code> routine</td>
</tr>
<tr>
<td>FILL_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>FILL_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>FILL_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>FILL_DATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>FILL_ENV_WITH_Sweep</td>
<td>the modules <code>fill_data_envelope</code> and <code>sweep_data</code> cannot be used interchangeably for the same data key, extension, version combination</td>
</tr>
<tr>
<td>BAD_VFMT</td>
<td>bad format character for variable width bin spacing</td>
</tr>
<tr>
<td>PHI_DIFF_UNITS</td>
<td>the sensors being processed do not process the same number of data levels (units)</td>
</tr>
<tr>
<td>FILL_PHI_FIRST</td>
<td>the starting azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td>FILL_PHI_LAST</td>
<td>the ending azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td>NO_EMPTY_BUFFERS</td>
<td>no spare buffers for data accumulation</td>
</tr>
</tbody>
</table>

Error codes returned by: `read_drec()`  
Error codes returned by: `convert_to_units()`  
Error codes returned by: `fill_sensor_info()`

ALL_OKAY  
routine terminated successfully

**DESCRIPTION**

**`Fill_data_envelope`** is the IDFS data read routine which returns the minimum and maximum data values encountered for the time duration being processed. **`Fill_data_envelope`** retrieves data for all sensors that return data for the time duration being processed. The data set of interest is referenced through the key value `data_key` which can be created using the `get_data_key` module. **`Fill_data_envelope`** processes sensor-specific data only, that is, it processes sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. Currently, there is no similar module defined to return the data envelope encountered for instrument status (mode) data.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the
IDFS routines will use the file descriptors defined for the version number specified. The user should call the `get_version_number` routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter `idf_data_ptr` is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the `create_idf_data_structure` routine is made. The user also has the option of calling the module `create_data_structure`, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the `create_idf_data_structure` routine N times to create N instances of the `idf_data` structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

This routine will process data one sweep at a time, placing the data into buffers which hold data that is accumulated over a specified time interval. Once the time interval has been processed, the routine will return the data buffers and a status value for each buffer which indicates when the buffer is ready for the user to retrieve. The user **must** call the module `set_time_values` before the `fill_data_envelope` module can be called since the `set_time_values` module is used to specify the base time value and reference location and the time interval (delta) to use to accumulate the data. If the `fill_data_envelope` routine determines that the `set_time_values` module has not been called, an error code is returned to the user.

Along with the data envelope being returned, there is a starting location and an ending location that is returned for each of the data buffers. The user may use these values as references to the base location specified in the call to the `set_time_values` module. That is, given a base time value, a time interval and a reference location, the `fill_data_envelope` routine will return the location of data with respect to time. The user may choose to ignore these values or may use these locations to plot data along an axis that is scaled with respect to time.
There are a constant number of data buffers that are used by the fill_data_envelope module. This number is defined as NUM_BUFFERS in the user_defs.h file. This file is described in section 1H of the IDFS Programmers Manual. These data buffers are utilized in a cyclic nature, with buffer 0 being re-used once buffer NUM_BUFFERS-1 has been filled. The data buffers that are ready to be processed are flagged with the status value BUFFER_READY. For each buffer, there are N many sub-buffers which hold the data in each of the requested data levels or units. The user must process the data contained within these buffers before the next call to the fill_data_envelope routine is made since the module will clear out these buffers for re-use. This holds true even when an LOS_STATUS or NEXT_FILE_STATUS status code is returned. Since the buffers are cyclic, the user may wish to keep a variable indicating the last buffer number processed so that the user can start at the time sample left off from the previous call to the fill_data_envelope module at the next call. It is important to note that there is one status flag per data buffer that is used by all sensors. If the sensors rotate or alternate when data is returned, the result may be that a buffer is flagged as BUFFER_READY but may not contain any data since the data buffers are reset or cleared out upon each call to the fill_data_envelope module. The user is advised to check the value or values in the bin_stat array. If all values are 0, no data was placed into the buffers.

The size and spacing of the data buffers are either defined by the user or by elements contained within the virtual instrument definition document. The user must call the set_bin_info module before calling the fill_data_envelope routine in order to specify how the binning of the data is to occur. If the fill_data_envelope routine determines that no binning scheme has been selected, an error code is returned to the user.

The default mode for the fill_data_envelope routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the fill_sensor_info module prior to calling the fill_data_envelope module. The user should make one call to the fill_sensor_info module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

Unlike the module fill_data, fill_data_envelope has no provisions for the averaging of the data returned since the data source must be a scalar source. In other words, there is no allowance for the collapsing of the data over data dimensions (refer to setCollapse_info if this statement is unclear). In addition, since the data source must be a scalar source, there is only one bin defined per data buffer; therefore, data is either returned or not returned. There are no provisions made to fill in missing bins since there is only one bin defined (refer to buffer_bin_fill if this statement is unclear).

The parameter exclude_dqual holds a single value that is compared against the d_qual value found in the header record for the sensor being processed. If the user wishes to exclude data that is flagged with a specific d_qual value, the user should set the exclude_dqual value to this specific value. If the user wishes to include all data encountered, the user should set the exclude_dqual value to 255.
ERRORS
All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
read_drec 1R
convert_to_units 1R
set_time_values 2R
set_bin_info 2R
fill_sensor_info 2R
get_data_key 1R
get_version_number 1R
create_data_structure 1R
create_idf_data_structure 1R
ret_codes 1H
user_defs 1H
libtrec_idfs 2H

BUGS
None

EXAMPLES
Obtain the data envelope for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_min, *ret_max;
SDDAS_LONG *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
SDDAS_LONG *bpix, *epix;
SDDAS_SHORT *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change, *buf_stat;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
```
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = fill_data_envelope (data_key, "", vnum, idf_data_ptr, &sen_numbers,
    &ret_min, &ret_max, &ret_bin, &bpix, &epix, &buf_stat,
    &num_sen, &num_units, &data_block, &start_time_yr,
    &start_time_day, &start_time_sec, &start_time_nano,
    &end_time_yr, &end_time_day, &end_time_sec, &end_time_nano,
    &hdr_change, 255);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by fill_data_envelope routine.\n", status);
    exit (-1);
}
**FILL_DISCONTINUOUS_DATA**

function - returns time-averaged data buffers for data sets that roll over to a minimum value when the maximum threshold has been reached

**SYNOPSIS**

```
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT fill_discontinuous_data (SDDAS_ULONG data_key,
SDDAS_CHAR *exten, SDDAS_USHORT version,
void *idf_data_ptr, SDDAS_SHORT **ret_sensors,
SDDAS_FLOAT **ret_data, SDDAS_FLOAT **ret_frac,
SDDAS_CHAR **bin_stat, SDDAS_LONG **bpix,
SDDAS_LONG **epix, SDDAS_CHAR **ret_stat,
SDDAS_SHORT *num_sen, SDDAS_SHORT **num_units,
SDDAS_SHORT *block_size, SDDAS_SHORT **stime_yr,
SDDAS_SHORT **stime_day, SDDAS_LONG **stime_sec,
SDDAS_LONG **stime_nano, SDDAS_SHORT **etime_yr,
SDDAS_SHORT **etime_day, SDDAS_LONG **etime_sec,
SDDAS_LONG **etime_nano, SDDAS_CHAR *hdr_change,
SDDAS_UCHAR exclude_dqual)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **idf_data_ptr** - pointer to the *idf_data* structure that temporarily holds sensor data and pertinent ancillary data for the data set of interest
- **ret_sensors** - an array which holds the sensor number(s) for which data is returned
  - the array is initialized to -1 in all elements; valid sensor numbers start with 0
- **ret_data** - pointer to the data being returned (data for all sensors processed)
- **ret_frac** - pointer to the normalization factors for the data being returned
- **bin_stat** - pointer to status flags which are associated with each data bin returned
  - 0 - no data has been placed into the data bin being processed
fill_discontinuous_data (2R)

1 - data has been placed into the data bin being processed

bpix - pointer to the starting pixel location for the data buffers returned

epix - pointer to the ending pixel location for the data buffers returned

ret_stat - pointer to the status of each of the data buffers being returned

UNTOUCHED_BUFFER - no data has ever been placed into the buffer

FREE_BUFFER - no data has been placed into the buffer being processed (ready for re-use)

PARTIAL_WORKING - data is being acquired into the buffer but is not ready for processing

BUFFER_READY - data has been acquired into the buffer and is ready for processing

num_sen - the number of elements in the ret_sensors array

num_units - an array holding the number of data sets to bypass in order to get to the data for the sensor being processed

block_size - the number of data values returned in a data buffer

stime_yr - pointer to the start time year values for the data buffers returned

stime_day - pointer to the start time day of year values for the data buffers returned

stime_sec - pointer to the start time of day values (in seconds) for the data buffers returned

stime_nano - pointer to the start time of day residuals (in nanoseconds) for the data buffers returned

etime_yr - pointer to the end time year values for the data buffers returned

etime_day - pointer to the end time day of year values for the data buffers returned

etime_sec - pointer to the end time of day values (in seconds) for the data buffers returned

etime_nano - pointer to the end time of day residuals (in nanoseconds) for the data buffers returned
hdr_change - flag which indicates a header change occurred while 
processing the data
  0 - a header change was not encountered 
during the processing of the data
  1 - a header change was encountered 
during the processing of the data
exclude_dqual - data is to be excluded if the d_qual flag associated 
with the data is set to the value specified
fill_discontinuous_data - routine status (see TABLE 1)

**TABLE 1.** Status Code Returned for **FILL_DISCONTINUOUS_DATA**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_DISC_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call <strong>file_open</strong> for this combination)</td>
</tr>
<tr>
<td>FILL_DISC_BASE_TIME_MISSING</td>
<td>the time interval information has not been set (user did not call <strong>set_time_values</strong> for this combination)</td>
</tr>
<tr>
<td>FILL_DISC_BIN_MISSING</td>
<td>The data binning information has not been allocated (user did not call <strong>set_bin_info</strong> for this combination)</td>
</tr>
<tr>
<td>FILL_DISC_CENTER_BAND_MISSING</td>
<td>the routine <strong>center_and_band_values</strong> has not been called prior to calling the <strong>fill_discontinuous_data</strong> routine</td>
</tr>
<tr>
<td>FILL_DISC_NO_PHI</td>
<td>data sets with PHI, MASS and/or CHARGE dimensions are not supported</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which hold information pertinent to the time-averaged data</td>
</tr>
<tr>
<td>FILL_DISC_MALLOC</td>
<td>no memory for structure which hold information pertinent to discontinuous data sets</td>
</tr>
<tr>
<td>FILL_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>FILL_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>FILL_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>FILL_DATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>SWEEP_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SWEEP_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SWEEP_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SWEEP_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>SWEEP_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>SWEEP_DATA_MALLOC</td>
<td>no memory for data buffer</td>
</tr>
<tr>
<td>DISC_DATA_MALLOC</td>
<td>no memory for the internal data buffers that are pertinent only to discontinuous data sets</td>
</tr>
<tr>
<td>FILL_WITH_SWEEP_DISC</td>
<td>the modules <strong>fill_discontinuous_data and sweep_discontinuous_data</strong> cannot be used interchangeably for the same data key, extension, version combination.</td>
</tr>
<tr>
<td>BAD_VFMT</td>
<td>bad format character for variable width bin spacing</td>
</tr>
<tr>
<td>DISC_TMP_MALLOC</td>
<td>no memory for scratch space utilized to process discontinuous data sets</td>
</tr>
<tr>
<td>NO_EMPTY_BUFFERS</td>
<td>no spare buffers for data accumulation</td>
</tr>
</tbody>
</table>

Error codes returned by **read_drec ()**

Error codes returned by **convert_to_units ()**

Error codes returned by **fill_sensor_info ()**
**DESCRIPTION**

*Fill_discontinuous_data* is the IDFS time-averaging data read routine, retrieving data for all sensors that return data for the time duration being processed. The data set of interest is referenced through the key value `data_key` which can be created using the `get_data_key` module. *Fill_discontinuous_data* processes sensor-specific data only, that is, it processes sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. If the instrument status (mode) data is desired, the user should use the `fill_mode_data` routine. *Fill_discontinuous_data* assumes that the data set of interest rolls over to a minimum value when the maximum threshold has been reached or to a maximum value when the minimum threshold has been reached. This assumption is crucial since multiple samples may be averaged together in a single buffer. Before each sample is added to the buffer, a check is made to see if a "boundary" or threshold has been crossed. If so, the value is adjusted so that the addition of the values result in a correct averaged value. Currently, these threshold values are preset at -180 (minimum threshold) and 180 (maximum threshold). If the data set does not roll over, the user should use the `fill_data` routine. If the data set of interest is a combination of roll over and non-roll over data, for example, longitude data being returned along with science data, the user may use the `fill_discontinuous_data` module in conjunction with the `fill_data` routine, using the `fill_data` routine to return the non-roll over data values and using the `fill_discontinuous_data` routine to return the roll over data values. In order to do this correctly, the user must make use of multiple version numbers so that the same data files can be opened more than once. That is, use one version number for the non-roll over data and another version number for the roll over data. All IDFS routines that utilize a version number must be called once for each unique version number.

The parameter `version` allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the `get_version_number` routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter `idf_data_ptr` is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the `create_idf_data_structure` routine is made. The user also has the option of calling the module `create_data_structure`, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the `create_idf_data_structure` routine N times to create N
instances of the \texttt{idf\_data} structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the \texttt{file\_open} routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable \texttt{exten}. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER\_DATA, which is set by the user, or in the user's home directory if the environment variable USER\_DATA is not set. To open the default IDFS data files, \texttt{exten} should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set \texttt{exten} to a null string for real-time scenarios.

This routine will process data one sweep at a time, placing the data into buffers which hold data that is accumulated over a specified time interval. Once the time interval has been processed, the routine will return the data buffers and a status value for each buffer which indicates when the buffer is ready for the user to retrieve. The user \textbf{must} call the module \texttt{set\_time\_values} before the \texttt{fill\_discontinuous\_data} module can be called since the \texttt{set\_time\_values} module is used to specify the base time value and reference location and the time interval (delta) to use to accumulate the data. If the \texttt{fill\_discontinuous\_data} routine determines that the \texttt{set\_time\_values} module has not been called, an error code is returned to the user.

Along with the data being returned, there is a starting location and an ending location that is returned for each of the data buffers. The user may use these values as references to the base location specified in the call to the \texttt{set\_time\_values} module. That is, given a base time value, a time interval and a reference location, the \texttt{fill\_discontinuous\_data} routine will return the location of data with respect to time. The user may chose to ignore these values or may use these locations to plot data along an axis that is scaled with respect to time.

There are a constant number of data buffers that are used by the \texttt{fill\_discontinuous\_data} module. This number is defined as \texttt{NUM\_BUFFERS} in the \texttt{user\_defs.h} file. This file is described in section 1H of the IDFS Programmers Manual. These data buffers are utilized in a cyclic nature, with buffer 0 being re-used once buffer \texttt{NUM\_BUFFERS-1} has been filled. The data buffers that are ready to be processed are flagged with the status value \texttt{BUFFER\_READY}. For each buffer, there are \texttt{N} many sub-buffers which hold the data in each of the requested data levels or units. The user must process the data contained within these buffers before the next call to the \texttt{fill\_discontinuous\_data} routine is made since the module will clear out these buffers for re-use. This holds true even when an \texttt{LOS\_STATUS} or \texttt{NEXT\_FILE\_STATUS} status code is returned. The data values must be normalized using the normalization factors returned along with the data. Since the buffers are cyclic, the user may wish to keep a variable indicating the last buffer number processed so that the user can start at the time sample left off from the previous call to the \texttt{fill\_discontinuous\_data} routine at the next call. It is important to note that there is one status flag per data buffer that is used by all sensors. If the sensors rotate or alternate when
data is returned, the result may be that a buffer is flagged as BUFFER_READY but may not contain any data since the data buffers are reset or cleared out upon each call to the `fill_discontinuous_data` module. The user is advised to check the value or values in the `bin_stat` array. If all values are 0, no data was placed into the buffers.

The size and spacing of the data buffers are either defined by the user or by elements contained within the virtual instrument definition document. The user must call the `set_bin_info` module before calling the `fill_discontinuous_data` routine in order to specify how the binning of the data is to occur. In addition, the user must call the `center_and_band_values` module before calling the `fill_discontinuous_data` module. If the `fill_discontinuous_data` routine determines that no binning scheme has been selected, an error code is returned to the user.

The user should be aware that the data buffers that come back from the `fill_discontinuous_data` module are NOT modified as far as missing bins is concerned. If the user wishes to fill in the missing bins according to the method specified in the call to the `set_bin_info` routine, the user must call the module `buffer_bin_fill`. If the data are collapsed over specified dimensions, the `buffer_bin_fill` module need not be called. The user should be advised that the `fill_discontinuous_data` routine can not process data sets with a PHI, MASS and/or CHARGE dimensionality.

The default mode for the `fill_discontinuous_data` routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the `fill_sensor_info` module prior to calling the `fill_discontinuous_data` module. The user should make one call to the `fill_sensor_info` module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

The parameter `exclude_dqual` holds a single value that is compared against the d_qual value found in the header record for the sensor being processed. If the user wishes to exclude data that is flagged with a specific d_qual value, the user should set the `exclude_dqual` value to this specific value. If the user wishes to include all data encountered, the user should set the `exclude_dqual` value to 255.

**ERRORS**
All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**
- `file_open` 1R
- `read_drec` 1R
- `convert_to_units` 1R
- `set_time_values` 2R
fill_discontinuous_data (2R)

fill_data 2R
fill_mode_data 2R
set_bin_info 2R
center_and_band_values 2R
fill_sensor_info 2R
buffer_bin_fill 2R
get_data_key 1R
get_version_number 1R
create_data_structure 1R
create_idf_data_structure 1R
ret_codes 1H
user_defsf 1H
libtrec_idfs 2H

BUGS
None

EXAMPLES
Obtain time-averaged data from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG *start_time_sec, *start_timeNano, *end_time_sec, *end_timeNano;
SDDAS_LONG *bpix, *epix;
SDDAS_SHORT *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change, *buf_stat;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);
status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n",
        status);
    exit (-1);
}

status = fill_discontinuous_data (data_key, ",", vnum, idf_data_ptr, &sen_numbers,
        &ret_data, &ret_frac, &ret_bin, &bpix, &epix, &buf_stat,
        &num_sen, &num_units, &data_block, &start_time_yr,
        &start_time_day,&start_time_sec, &start_time_nsec, &end_time_yr,
        &end_time_day,&end_time_sec, &end_time_nsec, &hdr_change,
        255);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by fill_discontinuous_data routine.\n", status);
    exit (-1);
}
FILL_MODE_DATA
function - returns time-averaged buffers for instrument status (mode) data

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT fill_mode_data (SDDAS_ULONG data_key,
                           SDDAS_CHAR *exten, SDDAS_USHORT version,
                           void *idf_data_ptr, SDDAS_SHORT **ret_modes,
                           SDDAS_FLOAT **ret_data, SDDAS_FLOAT **ret_frac,
                           SDDAS_CHAR **bin_stat, SDDAS_LONG **bpix,
                           SDDAS_LONG **epix, SDDAS_CHAR **ret_stat,
                           SDDAS_SHORT *num_modes,  SDDAS_SHORT **num_units,
                           SDDAS_SHORT *block_size,  SDDAS_SHORT **stime_yr,
                           SDDAS_SHORT **stime_day, SDDAS_LONG **stime_sec,
                           SDDAS_LONG **stime_nano, SDDAS_SHORT **etime_yr,
                           SDDAS_SHORT **etime_day, SDDAS_LONG **etime_sec,
                           SDDAS_LONG **etime_nano, SDDAS_CHAR *hdr_change,
                           SDDAS_UCHAR exclude_dqual)

ARGUMENTS
    data_key  - unique value which indicates the data set of interest
    exten   - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
    version   - IDFS data set identification number which allows for
multiple openings of the same data set
    idf_data_ptr  - pointer to the idf_data structure that temporarily holds
sensor data and pertinent ancillary data for the data set of
interest
    ret_modes  - an array which holds the instrument status (mode) bytes for
which data is returned
                 - the array is initialized to -1 in all elements;
                 valid mode numbers start with 0
    ret_data  - pointer to the data being returned (data for all modes
processed)
    ret_frac  - pointer to the normalization factors for the data being returned
    bin_stat - pointer to status flags which are associated with each data bin
returned
                 0 - no data has been placed into the data bin being
processed
                 1 - data has been placed into the data bin being
processed
    bpix   - pointer to the starting pixel location for the data buffers
returned
epix - pointer to the ending pixel location for the data buffers returned

ret_stat - pointer to the status of each of the data buffers being returned

  UNTOUCHED_BUFFER - no data has ever been placed into the buffer

  FREE_BUFFER - no data has been placed into the buffer being processed (ready for re-use)

  PARTIAL_WORKING - data is being acquired into the buffer but is not ready for processing

  BUFFER_READY - data has been acquired into the buffer and is ready for processing

num_modes - the number of elements in the ret_modes array

num_units - an array holding the number of data sets to bypass in order to get to the data for the instrument status (mode) value being processed

block_size - the number of data values returned in a data buffer

stime_yr - pointer to the start time year values for the data buffers returned

stime_day - pointer to the start time day of year values for the data buffers returned

stime_sec - pointer to the start time of day values (in seconds) for the data buffers returned

stime_nano - pointer to the start time of day residuals (in nanoseconds) for the data buffers returned

etime_yr - pointer to the end time year values for the data buffers returned

etime_day - pointer to the end time day of year values for the data buffers returned

etime_sec - pointer to the end time of day values (in seconds) for the data buffers returned

etime_nano - pointer to the end time of day residuals (in nanoseconds) for the data buffers returned

hdr_change - flag which indicates a header change occurred while processing the data

  0 - a header change was not encountered during the processing of the data

  1 - a header change was encountered during the processing of the data

exclude_dqual - data is to be excluded if the d_qual flag associated with the data is set to the value specified

fill_mode_data - routine status (see TABLE 1)
TABLE 1. Status Codes Returned for FILL_MODE_DATA

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_MODE_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>FILL_MODE_FILE_OPEN</td>
<td>the user did not request mode data processing when file_open was called</td>
</tr>
<tr>
<td>FILL_MODE_INFO_DUP</td>
<td>the requested data_key, exten, version combination has no memory allocated for the instrument status information</td>
</tr>
<tr>
<td>MODES_NOT_REQUESTED</td>
<td>the user did not call fill_mode_info for this combination</td>
</tr>
<tr>
<td>FILL_MODE_BASE_TIME_MISSING</td>
<td>the time interval information has not been set (user did not call set_time_values for this combination)</td>
</tr>
<tr>
<td>FILL_MODE_ARRAY_MALLOC</td>
<td>no memory for structure which hold information pertinent to the time-averaged data</td>
</tr>
<tr>
<td>ALLOC_MODE_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>MODE_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>MODE_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>MODE_DATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>FILL_WITH_SWEEP_MODE</td>
<td>The modules fill_mode_data and sweep_mode_data cannot be used interchangeably for the same data key, extension, version combination</td>
</tr>
<tr>
<td>NO_EMPTY_BUFFERS</td>
<td>no spare buffers for data accumulation</td>
</tr>
<tr>
<td>Error codes returned by read_drec ()</td>
<td></td>
</tr>
<tr>
<td>Error codes returned by convert_to_units ()</td>
<td></td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Fill_mode_data is the IDFS time-averaging data read routine, retrieving instrument status (mode) data for the time duration being processed. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. Fill_mode_data processes instrument status data only. If sensor-specific data is desired, that is, sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential data and / or background data, the user should use the fill_data / fill_discontinuous_data routine(s).

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter idf_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the create_idf_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is
sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_idf_data_structure routine \( N \) times to create \( N \) instances of the idf_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable SER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

This routine will process data one sweep at a time, placing the data into buffers which hold data that is accumulated over a specified time interval. Once the time interval has been processed, the routine will return the data buffers and a status value for each buffer which indicates when the buffer is ready for the user to retrieve. The user must call the module set_time_values before the fill_mode_data module can be called since the set_time_values module is used to specify the base time value and reference location and the time interval (delta) to use to accumulate the data. If the fill_mode_data routine determines that the set_time_values module has not been called, an error code is returned to the user.

Along with the data being returned, there is a starting location and an ending location that is returned for each of the data buffers. The user may use these values as references to the base location specified in the call to the set_time_values module. That is, given a base time value, a time interval and a reference location, the fill_mode_data routine will return the location of data with respect to time. The user may chose to ignore these values or may use these locations to plot data along an axis that is scaled with respect to time.

There are a constant number of data buffers that are used by the fill_mode_data module. This number is defined as NUM_BUFFERS in the user_defs.h file. This file is described in section 1H of the IDFS Programmers Manual. These data buffers are utilized in a cyclic nature, with buffer 0 being re-used once buffer NUM_BUFFERS-1 has been filled. The data buffers that are ready to be processed are flagged with the status value BUFFER_READY. For each buffer, there are \( N \) many sub-buffers which hold the data in each of the requested data levels or units. The user must process the data contained within these buffers before the next call to the fill_mode_data routine is made since the module will clear out these buffers for re-use. This holds true even when an LOS_STATUS or NEXT_FILE_STATUS status code is returned. The data values must be normalized using the normalization factors returned along with the data. Since the buffers are cyclic, the user may wish to keep a variable indicating the last buffer number processed so that the user can start at the time sample left off from the previous call to the fill_mode_data routine at the

238 December 28, 2012
next call. It is important to note that there is one status flag per data buffer that is used by all instrument status values.

In order to utilize the `fill_mode_data` routine, the user must select the units to be returned and the data cutoff values to be applied by calling the `fill_mode_info` module prior to calling the `fill_mode_data` module. The user should make one call to the `fill_mode_info` module for each instrument status byte that is to be retrieved for each units/data cutoff combination selected. If the `fill_mode_data` routine determines that the `fill_mode_info` module was never called, an error code is returned.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `read_drec` 1R
- `convert_to_units` 1R
- `set_time_values` 2R
- `fill_data` 2R
- `fill_discontinuous_data` 2R
- `fill_mode_info` 2R
- `get_data_key` 1R
- `get_version_number` 1R
- `mode_units_index` 1R
- `create_data_structure` 1R
- `create_idf_data_structure` 1R
- `ret_codes` 1H
- `user_defs` 1H
- `libtrec_idfs` 2H

**BUGS**

None

**EXAMPLES**

Obtain time-averaged instrument status values from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"
```
SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
SDDAS_LONG *bpix, *epix;
SDDAS_SHORT *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_SHORT status, *mode_numbers, num_modes, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change, *buf_stat;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = fill_mode_data (data_key, ",", vnum, idf_data_ptr, &mode_numbers,
&ret_data, &ret_frac, &ret_bin, &bpix, &epix, &buf_stat, &num_modes, &num_units, &data_block, &start_time_yr,
&start_time_day, &start_time_sec, &start_time_nano,
&end_time_yr, &end_time_day, &end_time_sec, &end_time_nano,
&hdr_change, 255);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by fill_mode_data routine.\n", status);
    exit (-1);
}
**FILL_MODE_INFO**

function - specifies the data cutoff values and units to be returned for the specified instrument status (mode) value

**SYNOPSIS**

```
#include "librec_idsf.h"
#include "ret_codes.h"

SDDAS_SHORT fill_mode_info (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                          SDDAS_USHORT version, SDDAS_SHORT mode_val,
                          SDDAS_FLOAT min, SDDAS_FLOAT max,
                          SDDAS_CHAR num_tbls,
                          SDDAS_CHAR *tbls_to_apply,
                          SDDAS_LONG *tbl_oper)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **mode_val** - the instrument status (mode) value of interest
- **min** - the lower cutoff value for data that are to be put into the data buffers, specified in terms of the units desired
- **max** - the upper cutoff value for data that are to be put into the data buffers, specified in terms of the units desired
- **num_tbls** - the number of elements specified in the `tbls_to_apply` and `tbl_oper` parameters
- **tbls_to_apply** - the tables that are to be applied in order to derive the desired units
- **tbl_oper** - the operations that are to be applied to the specified tables in order to derive the desired units
- **fill_mode_info** - routine status (see TABLE 1)

**TABLE 1. Status Codes Returned for FILL_MODE_INFO**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE_INFO_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call <code>file_open</code> for this combination)</td>
</tr>
<tr>
<td>MODE_FILE_OPEN</td>
<td>the user did not request mode data processing when <code>file_open</code> was called</td>
</tr>
<tr>
<td>MODE_INFO_DUP</td>
<td>the requested data_key, exten, version combination has no memory allocated for the instrument status information</td>
</tr>
<tr>
<td>MODE_INFO_NO_MODES</td>
<td>there are no instrument status (mode) values defined for the data set</td>
</tr>
<tr>
<td>MODE_INFO_MALLOC</td>
<td>no memory for structures which hold mode specific information</td>
</tr>
<tr>
<td>MODE_INFO_BASE_MALLOC</td>
<td>no memory for expansion of structures which hold mode specific information</td>
</tr>
<tr>
<td>MODE_INFO_BASE_REALLOC</td>
<td>no memory for expansion of the min/max values for the modes being processed</td>
</tr>
<tr>
<td>MODE_INFO_TBL_MALLOC</td>
<td>no memory for table number / table operation information</td>
</tr>
<tr>
<td>MODE_INFO_BASE_REALLOC</td>
<td>no memory for expansion of the min/max values for the modes being processed</td>
</tr>
</tbody>
</table>

241  December 28, 2012
STATUS CODE | EXPLANATION OF STATUS
--- | ---
ALL_OKAY | routine terminated successfully

**DESCRIPTION**

_**Fill_mode_info**_ is the IDFS routine that specifies which instrument status (mode) values are being returned by the _**fill_mode_data** / _**sweep_mode_data** module. The data set of interest is referenced through the key value _**data_key** which can be created using the _**get_data_key** module. The instrument status (mode) values are not sensor-specific, that is, they pertain to all sensors within the sensor set. If sensor-specific data is to be retrieved, the user should use the _**fill_sensor_info** routine to specify the data that is to be returned by the _**fill_data** / _**fill_discontinuous_data** / _**sweep_data** / _**sweep_discontinuous_data** / _**spin_data** / _**spin_data_pixel** modules. The _**fill_mode_info** module must be called _**prior** to calling the _**fill_mode_data** / _**sweep_mode_data** routine; otherwise, an error code will be returned. The user should make one call to the _**fill_mode_info** module for each instrument status value that is to be processed for each units/data cutoff combination selected.

The parameter _**version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the _**get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the _**file_open** routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable _**exten**. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable _**USER_DATA**, which is set by the user, or in the user's home directory if the environment variable _**USER_DATA** is not set. To open the default IDFS data files, _**exten** should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set _**exten** to a null string for real-time scenarios.

**ERRORS**

All errors within this routine are returned through the status variable. The include file _**ret_codes.h**_, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The _**ret_codes.h** file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- _**file_open**  1R
- _**fill_mode_data**  2R
BUGS
None

EXAMPLES
 Specify raw units, with cutoff values of 0 and 5 for instrument status byte 1 from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT mode_min, mode_max;
SDDAS_LONG *tbl_oper = NULL;
SDDAS_SHORT status, mode_val;
SDDAS_CHAR num_tbls, *tbls_to_apply = NULL;

mode_val = 1;
mode_min = 0;
mode_max = 5;
num_tbls = 0;
status = get_data_key("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = fill_mode_info (data_key, ",", vnum, mode_val, mode_min, mode_max,
                        num_tbls, tbls_to_apply, tbl_oper);
if (status != ALL_OKAY)
{
    printf("\n Error %d from fill_mode_info routine.\n", status);
    exit (-1);
}
FILL_SENSOR_INFO
    function - specifies the data cutoff values, data type and units to be returned for the
    specified sensor

SYNOPSIS
    #include "libtrec_idfs.h"
    #include "ret_codes.h"
    #include "user_defs.h"

    SDDAS_SHORT fill_sensor_info (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
    SDDAS_USHORT version, SDDAS_SHORT sensor,
    SDDAS_FLOAT min, SDDAS_FLOAT max,
    SDDAS_CHAR num_tbls, SDDAS_CHAR *tbls_to_apply,
    SDDAS_LONG *tbl_oper, SDDAS_CHAR data_type,
    SDDAS_CHAR cal_set)

ARGUMENTS
    data_key  - unique value which indicates the data set of interest
    exten     - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
    version   - IDFS data set identification number which allows for multiple
openings of the same data set
    sensor    - sensor identification number
    min       - the lower cutoff value for data that are to be put into the data
buffers, specified in terms of the units desired
    max       - the upper cutoff value for data that are to be put into the data
buffers, specified in terms of the units desired
    num_tbls  - the number of elements specified in the tbls_to_apply and
tbl_oper parameters
    tbls_to_apply - tables that are to be applied in order to derive the desired units
    tbl_oper  - the operations that are to be applied to the specified tables in
order to derive the desired units
    data_type - the type of data being requested
        1 - sensor data (SENSOR)
        2 - sweep step data (SWEEP_STEP)
        3 - calibration data (CAL_DATA)
        5 - data quality data (D_QUAL)
        6 - pitch angle data PITCH_ANGLE)
        7 - start azimuthal angle data
            (START_AZ_ANGLE)
        8 - stop azimuthal angle data
            (STOP_AZ_ANGLE)
        9 - spacecraft potential data
            (SC_POTENTIAL)
       10 - background data
            (BACKGROUND)
cal_set - the calibration set from which requested calibration data (CAL_DATA) is to be retrieved
- If calibration data is not being requested, this parameter is not utilized and it is suggested that the user pass a value of zero for this parameter.

fill_sensor_info - routine status (see TABLE 1)

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_SENSOR_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>FILL_SENSOR_MODE_TYPE</td>
<td>instrument status (mode) data is not supported by the fill_sensor_info routine</td>
</tr>
<tr>
<td>FILL_SENSOR_MALLOC</td>
<td>no memory for structures which hold sensor specific information</td>
</tr>
<tr>
<td>FILL_SENSOR_REALLOC</td>
<td>no memory for expansion of structures which hold sensor specific information</td>
</tr>
<tr>
<td>FILL_SENSOR_BASE_MALLOC</td>
<td>no memory for min/max values for the sensors being processed</td>
</tr>
<tr>
<td>FILL_SENSOR_BASE_REALLOC</td>
<td>no memory for expansion of the min/max values for the sensor being processed</td>
</tr>
<tr>
<td>FILL_SENSOR_TBL_MALLOC</td>
<td>no memory for table number / table operation information</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

**fill_sensor_info** is the IDFS routine that specifies what data is being returned by the IDFS routines that return time-averaged data (**fill_data** / **fill_discontinuous_data**), sample-averaged data (**sweep_data** / **sweep_discontinuous_data**) or spin-averaged data (**spin_data** / **spin_data_pixel**). The data set of interest is referenced through the key value **data_key** which can be created using the **get_data_key** module. The **fill_sensor_info** module should be used for sensor-specific data only, that is, for sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. If the instrument status (mode) data is to be retrieved, the user should use the **fill_mode_info** routine to specify which status bytes are to be returned by the **fill_mode_data** module.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIFD file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the **get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the **file_open** routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable **exten**. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files
must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

If it has been determined that the fill_sensor_info module has not been called, the default mode for the time-averaging, sample-averaging, and spin-averaging routines is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may specify the type of data, the units to be returned and the data cutoff values to be applied by calling the fill_sensor_info module prior to calling the time-averaging, sample-averaging, or spin-averaging module. The user should make one call to the fill_sensor_info module for each sensor that is to be processed for each data type/units/data cutoff combination selected.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
  file_open   1R
  fill_mode_info  2R
  fill_data  2R
  fill_discontinuous_data  2R
  sweep_data  2R
  sweep_discontinuous_data  2R
  spin_data  2R
  spin_data_pixel  2R
  get_data_key  1R
  get_version_number  1R
  ret_codes  1H
  user_defes  1H
  libtrec_idfs  2H

BUGS
None

EXAMPLES
Specify raw units, with cutoff values of 10 and 25 for SENSOR data for all defined sensors from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. Assume that there are 3 sensors applicable to this virtual instrument.
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
register SDDAS_SHORT sensor;
SDDAS_FLOAT sen_min, sen_max;
SDDAS_LONG *tbl_oper = NULL;
SDDAS_SHORT status;
SDDAS_CHAR data_type, num_tbls, *tbls_to_apply = NULL;

data_type = SENSOR;
sen_min = 10.0;
sen_max = 25.0;
num_tbls = 0;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

for (sensor = 0; sensor < 3; ++sensor)
{
    status = fill_sensor_info (data_key, ",", vnum, sensor, sen_min, sen_max,
        num_tbls, tbls_to_apply, tbl_oper, data_type, 0);
    if (status != ALL_OKAY)
    {
        printf ("\n Error %d from fill_sensor_info routine.\n", status);
        exit (-1);
    }
}
FILL_THETA_MATRIX
  function - fills in the data matrix that is used to collapse over the theta and scan dimensions

SYNOPSIS
  #include "libtrec_idfs.h"
  #include "ret_codes.h"
  #include "user defs.h"

  SDDAS_SHORT fill_theta_matrix (SDDAS_Ulong data_key, SDDAS_CHAR *exten,
                                SDDAS_SHORT version, SDDAS_SHORT num_sen,
                                SDDAS_SHORT *ret_sensors, SDDAS_FLOAT *ret_data,
                                SDDAS_FLOAT *ret_frac, SDDAS_CHAR *bin_stat,
                                SDDAS_SHORT *num_units, SDDAS_CHAR cur_buf,
                                SDDAS_SHORT sen_units)

ARGUMENTS
  data_key    - unique value which indicates the data set of interest
  exten      - two character extension to be added to IDFS file names when
               default files are not to be used, otherwise a null string
  version    - IDFS data set identification number which allows for multiple
               openings of the same data set
  num_sen    - the number of elements in the ret_sensors array
  ret_sensors - an array which holds the sensor number(s) for which data is
                 returned
                - the array is initialized to -1 in all elements; valid
                 sensor numbers start with 0
  ret_data   - pointer to the data being returned (data for all sensors
               processed)
  ret_frac   - pointer to the normalization factors for the data being returned
  bin_stat   - pointer to status flags which are associated with each data bin
               returned
                0    - no data has been placed into the data bin
                1    - data has been placed into the data bin being
                       processed
  num_units  - an array holding the number of data sets to bypass in order to
               get to the data for the sensor being processed
  cur_buf    - the current buffer being processed (number between 0 and
               NUM_BUFFERS-1)
  sen_units  - the number of units or data levels defined for the sensor in
               question
  fill_theta_matrix - routine status (see TABLE 1)
### TABLE 1. Status Code Returned for FILL_THETA_MATRIX

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_THETA_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>FILL_THETA_COLLAPSE</td>
<td>no memory has been allocated to hold the collapsing information (user did not call set_collapse_info for this combination)</td>
</tr>
<tr>
<td>THETA_DIFF_UNITS</td>
<td>the sensor do not process the same number of data levels</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

### DESCRIPTION

**Fill_theta_matrix** is the IDFS routine which assembles a data matrix when data is to be reduced in either the theta and/or scan dimensions. If the user is also collapsing the data over the charge, mass and/or phi dimensions, there is no need to call this module. The data matrix for data reduction will be acquired within the call to the time-averaging module (*fill_data / fill_discontinuous_data*), the sample-averaging module (*sweep_data / sweep_discontinuous_data*) or the spin-averaging module (*spin_data / spin_data_pixel*). The data set of interest is referenced through the key value *data_key* which can be created using the *get_data_key* module. This module can not be used in conjunction with the *fill_mode_data / sweep_mode_data* module since dimensionality is associated with sensor-specific data and instrument status data is not sensor-specific.

The parameter *version* allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the *get_version_number* routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the *file_open* routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable *exten*. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, *exten* should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set *exten* to a null string for real-time scenarios.

The time-averaging or sample-averaging routine will return a constant number of data buffers. This number is defined as NUM_BUFFERS in the *user_defs.h* file. This file is described in section 1H of the IDFS Programmers Manual. These data buffers are utilized in a cyclic nature, with buffer 0 being re-used once buffer NUM_BUFFERS-1 has been
filled. If the user is collapsing the data over the theta or scan dimension, the user must call this module in conjunction with the collapse_dimensions module, processing one buffer at a time. The user should process only those buffers that are flagged with the status value BUFFER_READY. All sensors that contain data in the specified buffer are processed, as well as each different data level or unit. Since the buffers are cyclic, the user may wish to keep a variable indicating the last buffer number processed so that the user can start at the time sample left off from the previous call to the time-averaging or sample-averaging routine at the next call.

The spin-averaging routine will return a single data buffer. If the user is collapsing the data over the theta or scan dimension, the user must call this module in conjunction with the collapse_dimensions module. All sensors that contain data in the specified buffer are processed, as well as each different data level or unit.

The call to the fill_theta_matrix routine should be called once for each buffer processed. The user may wish to collapse the data over different dimensions and ranges, in which case, there would be multiple calls to the collapse_dimensions module but there should be just one call to the fill_theta_matrix routine.

The parameter sen_units holds the number of units or data levels defined for the sensors. This value is used as an index to get to the first buffer returned by the time-averaging, sample-averaging, or spin-averaging routine for each sensor. The value for this parameter can be retrieved by calling the module units_index, with the next to the last argument from the end holding the value to be passed to this module. The parameters num_sen, ret_sensors, ret_data, ret_frac, bin_stat and num_units are returned from the call to the time-averaging, sample-averaging, or spin-averaging routine.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
- file_open 1R
- fill_data 2R
- fill_discontinuous_data 2R
- sweep_data 2R
- sweep_discontinuous_data 2R
- spin_data 2R
- spin_data_pixel 2R
- collapse_dimensions 2R
- setCollapseInfo 2R
- units_index 2R
- getDataKey 1R
- getVersionNumber 1R
In order to produce a line plot from the RTLA virtual instrument, data must be collapsed over a frequency range. Assuming that a single data level (raw units) is being returned, fill the data matrix used for data reduction and collapse the data over the two ranges for each buffer that is ready to be processed. The virtual instrument RTLA is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"
#define DUMMY_VAL 0

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_data, *ret_frac, *data_ptr, start[6], stop[6];
SDDAS_LONG *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
SDDAS_LONG *bpix, *epix;
SDDAS_SHORT *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change, *buf_stat, cur_buf, buf_num, dimen[6];
static SDDAS_CHAR which_buf = 0;
void *idf_data_ptr;

dimen[0] = DIMEN_ON;
start[0] = 0.16;
stop[0] = 0.9;
start[1] = stop[1] = 0.0;
start[2] = stop[2] = 0.0;
start[3] = stop[3] = 0.0;
start[4] = stop[4] = 0.0;
start[5] = stop[5] = 0.0;

status = get_data_key("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = fill_data (data_key, "", vnum, idf_data_ptr, &sen_numbers, &ret_data, &ret_frac,
                  &ret_bin, &bpix, &epix, &buf_stat, &num_sen, &num_units,
                  &data_block, &start_time_yr, &start_time_day, &start_time_sec,
                  &start_time_nano, &end_time_yr, &end_time_day, &end_time_sec,
                  &end_time_nano, &hdr_change, 255);
if (status >= 0)
{
    cur_buf = which_buf;
    for (buf_num = 0; buf_num < NUM_BUFFERS; ++buf_num)
    {
        if (*(buf_stat + cur_buf) == BUFFER_READY)
        {
            status = fill_theta_matrix (data_key, "", vnum, num_sen, sen_numbers, ret_data,
                                        ret_frac, ret_bin, num_units, cur_buf, 1);
            if (status != ALL_OKAY)
            {
                printf ("\n Error %d from fill_theta_matrix routine.\n", status);
                exit (-1);
            }
        }
    }
    status = collapse_dimensions (data_key, "", vnum, 1, dimen, start, stop,
                                 STRAIGHT_AVG, DUMMY_VAL, &data_ptr, 0, 1, 3,
                                 0.0, 1, 1, 0, 0, 1, cur_buf);
    if (status != ALL_OKAY)
    {
        printf ("\n Error %d from collapse_dimensions routine.\n", status);
        exit (-1);
    }
    printf ("\n data value = %f", *data_ptr);
}

start[0] = 0.3;
stop[0] = 0.6;
status = collapse_dimensions (data_key, "", vnum, 1, dimen, start, stop,
                             STRAIGHT_AVG, DUMMY_VAL, &data_ptr, 0, 1, 3,
                             0.0, 1, 1, 0, 1, 1, cur_buf);
if (status != ALL_OKAY)
{  
  printf("\n Error %d from collapse_dimensions routine.\n", status);  
  exit (-1);  
}
printf("\n data value = %f", *data_ptr);

    which_buf = (cur_buf + 1) % NUM_BUFFERS;
}  
cur_buf = (cur_buf + 1) % NUM_BUFFERS;
}  
}
MODE_UNITS_INDEX

function - returns index values to access the data returned by the fill_mode_data / sweep_mode_data module for the cutoff / units combination specified

SYNOPSIS

#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT mode_units_index (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, SDDAS_SHORT mode_val,
SDDAS_FLOAT min, SDDAS_FLOAT max,
SDDAS_CHAR num_tbls, SDDAS_CHAR *tbls_to_apply,
SDDAS_LONG *tbl_oper, SDDAS_SHORT *units_ind,
SDDAS_SHORT *num_units)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple openings of the same data set
mode_val - the instrument status (mode) value of interest
min - the lower cutoff value for data that are to be put into the data buffers, specified in terms of the units desired.
max - the upper cutoff value for data that are to be put into the data buffers, specified in terms of the units desired.
num_tbls - the number of elements specified in the tbls_to_apply and tbl_oper parameters
tbls_to_apply - the tables that are to be applied in order to derive the desired units
tbl_oper - the operations that are to be applied to the specified tables in order to derive the desired units
units_ind - index value returned to access the correct sub-buffer returned from the fill_mode_data / sweep_mode_data routine for the cutoff/units combination requested
num_units - the number of units or data levels defined for the instrument status (mode) value in question (used as an index to get to the first buffer returned by the fill_mode_data / sweep_mode_data routine for the mode in question)
mode_units_index - routine status (see TABLE 1)
TABLE 1. Status Codes Returned for MODE_UNITS_INDEX

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE_UNITS_IND_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>MODE_UNITS_FILE_OPEN</td>
<td>the user did not request mode data processing when file_open was called</td>
</tr>
<tr>
<td>MODE_UNITS_INFO_DUP</td>
<td>the requested data_key, exten, version combination has no memory allocated for the instrument status information</td>
</tr>
<tr>
<td>MODE_UNITS_NO_MODE</td>
<td>the requested instrument status (mode) value was not found amongst the defined cutoff/units combinations (user did not call fill_mode_info for this combination)</td>
</tr>
<tr>
<td>MODE_UNITS_NO_MATCH</td>
<td>the cutoff/units combination requested was not found for the specified instrument status (mode) value</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Mode_units_index is the IDFS routine that returns index values that are used to access the data buffers returned by the fill_mode_data / sweep_mode_data routine to retrieve the data for the instrument status (mode) value, cutoff/units combination specified. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The instrument status (mode) values are not sensor-specific, that is, they pertain to all sensors within the sensor set. If sensor-specific data is also being processed, the user should use the units_index routine to retrieve index values to access the data buffers returned by the fill_data / fill_discontinuous_data / sweep_data / sweep_discontinuous_data / spin_data / spin_data_pixel routines.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.
The user may elect to call the `mode_units_index` routine every time a return from the `fill_mode_data` / `sweep_mode_data` routine is made or may call the `mode_units_index` routine once for each instrument status (mode) value, cutoff/units combination requested and save the index values into variables for later usage. In either case, the call(s) to the `mode_units_index` routine must be made after ALL calls to the `fill_mode_info` routine have been made.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `fill_mode_data` 2R
- `sweep_mode_data` 2R
- `fill_mode_info` 2R
- `units_index` 2R
- `get_data_key` 1R
- `get_version_number` 1R
- `ret_codes` 1H
- `libtrec_idfs` 2H

**BUGS**

None

**EXAMPLES**

Retrieve the index values to access data that is returned for instrument status byte 1 from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. Assume that there is one table applicable and the cutoff values of 0 and 5 are to be used.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS ULONG data_key;
SDDAS USHORT vnum;
SDDAS FLOAT mode_min, mode_max;
SDDAS LONG tbl_oper[1];
SDDAS SHORT uuid_base, status, mode_units, mode_val;
SDDAS CHAR tbls_to_apply[1], num_tbls;

mode_min = 0.0;
mode_max = 5.0;
mode_val = 1;
```
num_tbls = 1;
tbls_to_apply[0] = 0;
tbl_oper[0] = 0; /* look-up operation */

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = mode_units_index (data_key, ",", vnum, mode_val, mode_min, mode_max,
        num_tbls, tbls_to_apply, tbl_oper, &uind_base,
        &mode_units);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by mode_units_index routine.\n", status);
    exit (-1);
}
NUMBER_OF_DATA_BINS
function - returns the number of data bins created for the data set of interest

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT number_of_data_bins (SDDAS_ULONG data_key, 
SDDAS_CHAR *exten, SDDAS_USHORT version, 
SDDAS_SHORT *num_bins)

ARGUMENTS
data_key - unique value which indicates the data set of interest
exten   - two character extension to be added to IDFS file
version - IDFS data set identification number which allows for
num_bins - the number of data bins created for the data set
number_of_data_bins - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for NUMBER_OF_DATA_BINS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBINS_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for</td>
</tr>
<tr>
<td></td>
<td>processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>NBINS_NO_BINS</td>
<td>no memory has been allocated to hold the binning information (user did not call</td>
</tr>
<tr>
<td></td>
<td>set_bin_info for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION
Number_of_data_bins is the IDFS routine that returns the number of data bins that are
utilized by the IDFS routines that return time-averaged data (fill_data and
fill_discontinuous_data), sample-averaged data (sweep_data and
sweep_discontinuous_data), or spin-averaged data (spin_data and spin_data_pixel). The
creation of the data bins is handled by the call to the set_bin_info module. If the
set_bin_info module has not been called, an error code is returned to the calling module.
The data set of interest is referenced through the key value data_key which can be created
using the get_data_key module.

The parameter version allows multiple file openings for an IDFS data set. If the data,
header and VIDF file for the specified data set need to be opened just once for processing,
the same version number should be passed to all IDFS routines. However, for multiple file
openings, the version number should be unique and all file manipulations performed by the
IDFS routines will use the file descriptors defined for the version number specified. The
user should call the get_version_number routine to retrieve a unique version number
instead of choosing this value themselves. The retrieval of multiple data parameters from a
A single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `fill_data` 2R
- `fill_discontinuous_data` 2R
- `sweep_data` 2R
- `sweep_discontinuous_data` 2R
- `spin_data` 2R
- `spin_data_pixel` 2R
- `set_bin_info` 2R
- `get_data_key` 1R
- `get_version_number` 1R
- `ret_codes` 1H
- `libtrec_idfs` 2H

**BUGS**

None

**EXAMPLES**

Determine the number of data bins created for the SPIA virtual instrument, which is part of the SPREE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. For this example, assume that the module `set_bin_info` has previously been called.
#include "libtree_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status, num_bins;

status = get_data_key("TSS", "TSS-1", "SPREE", "SPREE", "SPIA", &data_key);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);
.
.
status = number_of_data_bins(data_key, "", vnum, &num_bins);
if (status != ALL_OKAY)
{
    printf("\n Error %d from number_of_data_bins routine.\n", status);
    exit (-1);
}
NUMBER_OF_PHI_BINS

function - returns the number of phi bins created for the data set of interest

SYNOPSIS

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT number_of_phi_bins (SDDAS_ULONG data_key, 
        SDDAS_CHAR *exten, SDDAS_USHORT version, 
        SDDAS_SHORT *num_phi_bins)
```

ARGUMENTS

- **data_key**: unique value which indicates the data set of interest
- **exten**: two character extension to be added to IDFS file names when default files are not to be used, otherwise, a null string
- **version**: IDFS data set identification number which allows for multiple openings of the same data set
- **num_phi_bins**: the number of phi bins created for the data set
- **number_of_phi_bins**: routine status (see TABLE 1)

**TABLE 1.** Status Codes Returned for NUMBER_OF_PHI_BINS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPHI_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>NPHI_NO_BINS</td>
<td>no memory has been allocated to hold the collapsing information (user did not call set_collapse_info for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

*Number_of_phi_bins* is the IDFS routine that returns the number of phi bins that are utilized by the IDFS routines that return time-averaged data (*fill_data* and *fill_discontinuous_data*), sample-averaged data (*sweep_data* and *sweep_discontinuous_data*) or spin-averaged data (*spin_data* and *spin_data_pixel*). The creation of the phi bins is handled by the call to the *set_collapse_info* routine. If the *set_collapse_info* module has not been called, an error code is returned to the calling module. The data set of interest is referenced through the key value *data_key* which can be created using the *get_data_key* module.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the *get_version_number* routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a
single data source does not constitute the need for multiple version numbers; a single
version number will suffice.

If the **file_open** routine is not to open the default set of IDFS files but a modified set of
IDFS files, the two character extension applied to these data files must be supplied to this
routine within the string variable **exten**. These files must have the identical name as the
IDFS files with the two character identification code appended to the end of the file names
(i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files
must reside either in the directory specified by the environment variable USER_DATA,
which is set by the user, or in the user's home directory if the environment variable
USER_DATA is not set. To open the default IDFS data files, **exten** should be set to a null
string. The usage of modified data sets is limited to post acquisition data; therefore, it is
suggested that the user set **exten** to a null string for real-time scenarios.

**ERRORS**

All errors within this routine are returned through the status variable. The include file
**ret_codes.h**, which includes all possible return values, should be included so that the
mnemonics for the return codes can be referenced. The **ret_codes.h** file is described in
section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- **file_open**  
- **fill_data**  
- **fill_discontinuous_data**  
- **sweep_data**  
- **sweep_discontinuous_data**  
- **spin_data**  
- **spin_data_pixel**  
- **set_collapse_info**  
- **get_data_key**  
- **get_version_number**  
- **ret_codes**  
- **libtrec_idfs**

**BUGS**

None

**EXAMPLES**

Determine the number of phi bins created for the SPIA virtual instrument, which is part of
the SPREE instrument/experiment, which is part of the TSS-1 mission, which is identified
with the TSS project. For this example, assume that the module **set_collapse_info** has
previously been called.
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status, num_phi_bins;

status = get_data_key ("TSS", "TSS-1", "SPREE", "SPREE", "SPIA", &data_key);
if (status != ALL_OKAY)
{
    printf ("Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);
.
.

status = number_of_phi_bins (data_key, ",", vnum, &num_phi_bins);
if (status != ALL_OKAY)
{
    printf ("Error %d from number_of_phi_bins routine.\n", status);
    exit (-1);
}
RETURN_CENTER_AND_BAND_PTRS

function - returns the address of the center sweep and band width values associated with the data bins for the specified sensor

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT return_center_and_band_ptrs (SDDAS_ULONG data_key,
SDDAS_CHAR *exten, SDDAS_USHORT version,
SDDAS_SHORT sensor, SDDAS_FLOAT **center_ptr,
SDDAS_FLOAT **low_ptr, SDDAS_FLOAT **high_ptr)

ARGUMENTS
data_key   - unique value which indicates the data set of interest
exten    - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
version    - IDFS data set identification number which allows for multiple openings of the same data set
sensor    - sensor identification number
center_ptr   - pointer to the location that holds the center sweep values
low_ptr   - pointer to the location that holds the lower bands for non-contiguous bands or all band widths for contiguous bands
high_ptr   - pointer to the location that holds the upper bands for non-contiguous bands
return_center_and_band_ptrs - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for RETURN_CENTER_AND_BAND_PTRS

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET_CENTER_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>RET_CBPTR_NOT_FOUND</td>
<td>no memory has been allocated to hold the binning information (user did not call set_bin_info for this combination)</td>
</tr>
<tr>
<td>RET_CBPTR_NO_SENSOR</td>
<td>the specified sensor is not flagged as a sensor to be process for this data set</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Return_center_and_band_ptrs is the IDFS routine that returns pointers to the center and band width sweep step values for the specified sensor, which have been created by a call to the center_and_band_values module. For any given virtual instrument, there may be one set of sweep step values to be used by all sensors or there may be a set of sweep step values defined for each individual sensor. The sweep step values are used by the IDFS routines that return time-averaged data (fill_data / fill_discontinuous_data), sample-averaged data
(sweep_data / sweep_discontinuous_data) or spin-averaged data (spin_data / spin_data_pixel). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

The contents of the memory locations returned by this module should NOT be altered since the calculated center/band width values are used by the time-averaging, sample-averaging, or spin-averaging routine when processing the data. If the returned values need to be modified, for example, to take the log of the values, the user should allocate space to hold the values, copy the values into this space and modify the values there.

The module returns two possible pointers for the location(s) that hold the lower and upper band width values. In the case where the bands are non-contiguous, both the low_ptr and high_ptr will reference memory locations that hold the band width values. In the case where the bands are contiguous, there is no need to hold separate upper and lower values – the upper limit of the current band is the lower limit of the next band. In this case, one extra memory location is allocated, the high_ptr pointer is set to nil or 0 and low_ptr is set to reference the location that holds the band width values.

ERRORS

All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.
SEE ALSO

file_open  1R
fill_data  2R
fill_discontinuous_data  2R
sweep_data  2R
sweep_discontinuous_data  2R
spin_data  2R
spin_data_pixel  2R
set_bin_info  2R
center_and_band_values  2R
get_data_key  1R
get_version_number  1R
ret_codes  1H
libtrec_idfs  2H

BUGS

None

EXAMPLES

Print out the center sweep values used for binning the data for sensor 0 for the SPIA virtual instrument, which is part of the SPREE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. This code segment assumes that calls to set_bin_info and center_and_band_values modules have been made.

```
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
register SDDAS_SHORT bins, num_bins;
SDDAS_FLOAT *center_ptr, *low_ptr, *high_ptr;
SDDAS_SHORT status;

status = get_data_key ("TSS", "TSS-1", "SPREE", "SPREE", "SPIA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = number_of_data_bins (data_key, "", vnum, &num_bins);
if (status != ALL_OKAY)
{
    printf ("\n Error %d from number_of_data_bins routine.\n", status);
    exit (-1);
```
return_center_and_band_ptrs (2R)

    }

    status = return_center_and_band_ptrs (data_key, ",", vnum, 0, &center_ptr, &low_ptr, &high_ptr);
    if (status != ALL_OKAY)
    {
        printf ("\n Error %d from return_center_and_band_ptrs routine.\n", status);
        exit (-1);
    }

    for (bins = 0; bins < num_bins; ++bins)
        printf ("center sweep value [%d] = %.2f\n", bins, *(center_ptr + bins));
RETURN_PHI_PTRS

function - returns the phi bins and the data from the phi data matrix for the specified sensor for the specified buffer

SYNOPSIS

#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT return_phi_ptrs (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
   SDDAS_ULONG version, SDDAS_SHORT sensor,
   SDDAS_FLOAT **ret_data, SDDAS_CHAR cyclic,
   SDDAS_SHORT order, SDDAS_LONG need_filled,
   SDDAS_FLOAT tension, SDDAS_CHAR bin_project,
   SDDAS_SHORT unit_index, SDDAS_SHORT *num_phi,
   SDDAS_FLOAT **phi_bins, SDDAS_SHORT *data_size,
   SDDAS_CHAR cur_buf)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exen - two character value which indicates the data set of interest
version - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
sensor - IDFS data set identification number which allows for multiple openings of the same data set
ret_data - sensor identification number
order - pointer to the data being returned
flag indicating if the data is cyclic
0 - data is not cyclic
1 - data is cyclic
order - the order of the fit, i.e. 1, 2, 3, etc.
- This parameter is used if the bin fill method chosen in the call to the set_bin_info routine is any value other than NO_BIN_FILL.
need_filled - the number of filled bins needed in order to fill in the missing data bins
tension - the weighting of the data
bin_project - flag indicating if the data is to be projected into empty bins beyond the first or last data bin which contains data
0 - do not project the data
1 - project the data
unit_index - index value specifying which data level or unit is to be returned for the sensor in question
num_phi - the number of phi bins that are being returned
phi_bins - pointer to the location that holds the phi bin values
data_size - the number of data values returned per phi bin
cur_buf - the current buffer being processed (number between 0 and NUM_BUFFERS-1)
Return_phi_ptrs is the IDFS routine that returns both the phi band limits and the corresponding sensor data for a given sensor. The sensor data is acquired by the IDFS routines that return time-averaged data (fill_data), sample-averaged data (sweep_data), or spin-averaged data (spin_data / spin_data_pixel). The width of the phi bins was created using the information specified by the call to the set_collapse_info module. If the set_collapse_info module has not been called or if the phi bins were disabled by the call to the set_collapse_info module, an error code is returned to the calling module. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

The pointer ret_data references the data for the sensor requested. The data is laid down by phi bin, with data_size elements being returned per phi bin. The data returned can be thought of as a 2-dimensional matrix, with data_size rows and num_phi columns. This
module must be called after a call to the \texttt{fill_data} / \texttt{sweep_data} / \texttt{spin_data} / \texttt{spin_data_pixel} routine, which fills in the phi data matrix, has been made. This module can not be utilized in conjunction with the \texttt{fill_discontinuous_data} / \texttt{sweep_discontinuous} module since the module can not process data sets with a PHI, MASS and/or CHARGE dimensionality. The user should process only those buffers that are flagged with the status value \texttt{BUFFER\_READY}.

**ERRORS**

All errors within this routine are returned through the status variable. The include file \texttt{ret_codes.h}, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The \texttt{ret_codes.h} file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

\begin{description}
\item[file_open] 1R
\item[fill\_data] 2R
\item[sweep\_data] 2R
\item[spin\_data] 2R
\item[spin\_data\_pixel] 2R
\item[set\_collapse\_info] 2R
\item[get\_data\_key] 1R
\item[get\_version\_number] 1R
\item[ret\_codes] 1H
\item[libtrec\_idfs] 2H
\end{description}

**BUGS**

None

**EXAMPLES**

Print out each phi band and associated data values for sensor 0 for the SPIA virtual instrument, which is part of the SPREE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. For this example, assume that only one data level or unit is returned by the \texttt{fill\_data} module (default mode) and that \texttt{buf\_stat} had been set.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS\_ULONG data\_key;
SDDAS\_USHORT vnum;
register SDDAS\_SHORT phi, sample;
SDDAS\_FLOAT *data\_ptr, *phi\_bands, *data;
SDDAS\_SHORT status, phi\_bins, data\_block;
SDDAS\_CHAR cur\_buf, buf\_num, *buf\_stat;
static SDDAS\_CHAR which\_buf = 0;
```
status = get_data_key ("TSS", "TSS-1", "SPREE", "SPREE", "SPIA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);
cur_buf = which_buf;
for (buf_num = 0; buf_num < NUM_BUFFERS; ++buf_num)
{
    if (*(buf_stat + cur_buf) == BUFFER_READY)
    {
        status = return_phi_ptrs (data_key, ",", vnum, 0, &data_ptr, 0, 0, 2, 0.0,
                                0, 0, &phi_bins, &phi_bands, &data_block, cur_buf);
        if (status != ALL_OKAY)
        {
            printf ("\n Error %d returned by return_phi_ptrs routine.\n", status);
            exit (-1);
        }

        for (phi = 0; phi < phi_bins; ++phi)
        {
            printf ("phi bin is from %.2f to %.2f\n", *(phi_bands + phi),
                    *(phi_bands + phi + 1));
            data = data_ptr + phi * data_block;
            for (sample = 0; sample < data_block; ++sample)
                printf ("data[%d] = %.2e\n", sample, *(data + sample));
        }
        which_buf = (cur_buf + 1) % NUM_BUFFERS;
    }
    cur_buf = (cur_buf + 1) % NUM_BUFFERS;
}
SET_BIN_INFO

function - specifies how time-averaged, sample-averaged, or spin-averaged data is to be binned

SYNOPSIS

#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT set_bin_info (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                           SDDAS_USHORT version, SDDAS_CHAR swp_type,
                           SDDAS_FLOAT start, SDDAS_FLOAT stop,
                           SDDAS_FLOAT delta, SDDAS_SHORT num_bins,
                           SDDAS_CHAR swp_fmt, SDDAS_CHAR num_center,
                           SDDAS_CHAR *center_tbls, SDDAS_LONG *center_opers,
                           SDDAS_CHAR num_band, SDDAS_CHAR *band_tbls,
                           SDDAS_LONG *band_opers,
                           SDDAS_CHAR num_upper_band,
                           SDDAS_CHAR *upper_band_tbls,
                           SDDAS_LONG *upper_band_opers,
                           SDDAS_CHAR var_fmt, SDDAS_CHAR input_fmt,
                           SDDAS_CHAR bin_fill)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple
           openings of the same data set
swp_type - the format used to determine the number of data bins
            1 - use swp_len number of bins (FIXED_SWEEP)
            2 - user will specify the number of bin
                (VARIABLE_SWEEP)
start - the center value associated with the first bin for variable
        sweep processing
stop - the center value associated with the last bin for variable sweep
        processing
delta - the skip increment (delta) to use to create the bins for variable
        sweep processing
num_bins - the number of bins to create for variable sweep processing
swp_fmt - the spacing for the bins
          0 - use zero spacing (ZERO_SPACING)
          1 - use linear spacing (LIN_SPACING)
          2 - use logarithmic spacing (LOG_SPACING)
          3 - use variable width spacing
              (VARIABLE_SPACING)
num_center - the number of elements in the center_tbls and center_opers parameters

center_tbls - the tables that are to be applied to derive the units for the center sweep step values for variable width spaced bins

center_opers - the operations that are to be applied to the specified tables to derive the units for the center sweep step values for variable width spaced bins

num_band - the number of elements in the band_tbls and band_opers parameters

band_tbls - the tables that are to be applied to derive the units for the band width sweep step values for variable width spaced bins or the actual lower edge band width sweep step values for variable width spaced bins that use the ‘A’ format flag

band_opers - the operations that are to be applied to the specified tables to derive the units for the band width sweep step values for variable width spaced bins or the actual lower edge band width sweep step values for variable width spaced bins that use the ‘A’ format flag

num_upper_band - the number of elements in the upper_band_tbls and upper_band_opers parameters

upper_band_tbls - the tables that are to be applied to derive the units for the actual upper edge band width sweep step values for variable width spaced bins that use the ‘A’ format flag

upper_band_opers - the operations that are to be applied to the specified tables to derive the units for the actual upper edge band width sweep step values for variable width spaced bins that use the ‘A’ format flag

var_fmt - the format flag for variable width spacing

L or l - the center sweep values are used as the lower edge of the band width values

C or c - the center sweep values are used as the midpoints of the band width values

U or u - the center sweep values are used as the upper edge of the band width values

E or e - the center sweep values are used as the lower edge of the band width values and the scan widths specified are the actual upper edge of the band width values, not delta values

A or a - the actual center, lower edge band width and upper edge band width values are defined in the VIDF file (no computations off the center values are necessary)

input_fmt - the storage scheme for the binning of the data for variable sweep processing

1 - data is placed in the bin which encompasses the sweep value associated with the data
set_bin_info (2R) set_bin_info (2R)

277                  December 28, 2012
(POINT_STORAGE)

2 - data is placed in all bins which fully or partially contain the sweep range associated with the data (BAND_STORAGE)

bin_fill - method used to fill in any empty data bins
1 - leave the bins as is (NO_BIN.FILL)
2 - linearly interpolate across the row and then down the column of data (LIN_ROW.COL)
3 - linearly interpolate down the column and then across the row of data (LIN.COL.ROW)
4 - project data inward across the row and then down the column of data (CON.Row.COL)
5 - project data inward down the column and then across the row of data (CON.COL.ROW)
6 - apply a two-d least squares fit to the data (LEAST_SQ.FIT)

set_bin_info - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SET_BIN_INFO

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_BIN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>SETBIN_BAD_FMT</td>
<td>ZERO_SPACING or VARIABLE_SPACING can only be requested in conjunction with FIXED SWEEP processing</td>
</tr>
<tr>
<td>SET_BIN_MALLOC</td>
<td>no memory for data binning information</td>
</tr>
<tr>
<td>SET_BIN_INDEX_MALLOC</td>
<td>no memory for index values for the sensors</td>
</tr>
<tr>
<td>SET_VWIDTH_CENTER_MALLOC</td>
<td>no memory to hold center tables and operations for variable width spacing</td>
</tr>
<tr>
<td>SET_VWIDTH_BAND_MALLOC</td>
<td>no memory to hold band width tables and operations for variable width spacing</td>
</tr>
<tr>
<td>SET_VWIDTH_UPPER_BAND_MALLOC</td>
<td>no memory to hold upper band width tables and operations for variable width spacing</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>Error codes returned by set_collapse_info ()</td>
</tr>
</tbody>
</table>

DESCRIPTION

Set_bin_info is the IDFS routine that is utilized to define the size and spacing of the data buffers that will be filled by the IDFS routines that return time-averaged data (fill_data / fill_discontinuous_data), sample-averaged data (sweep_data / sweep_discontinuous_data), or spin-averaged data (spin_data / spin_data_pixel). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The first call to the set_bin_info routine for the data set specified will be used to generate the binning information. All subsequent calls with the identical data_key, exten and version parameters will be ignored. This module must be called prior to calling the time-averaging, sample-averaging, or spin-averaging routine; otherwise, an error code will be returned. If the only type of data to be processed for the
data set in question is instrument status (mode) data, the user does not need to call this module since the fill_mode_data / sweep_mode_data routine determines the size and spacing of the data buffers.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

There are two formats that can be used to bin the data, FIXED SWEEP and VARIABLE SWEEP. With a FIXED_SWEEP format, the bins are set up according to the information found in the VIDF file. The element swp_len is used to determine the number of bins. The data is stored into the bins by using the values found in the scan_index array as index values into the data bins. The scan_index array is contained in the header record. When specifying a FIXED sweep format, the values for the parameters start, stop, delta, num_bins and input_fmt are ignored. If the virtual instrument selected is a scalar instrument, the set_bin_info module will default to the FIXED SWEEP format with LINEAR SPACED bins, regardless of the setting of the parameters.

If the user selects a VARIABLE SWEEP format, the user must specify the number of bins to create (num_bins), the spacing of the bins (swp_fmt), the center value associated with the first bin (start), the center value associated with the last bin (stop), the skip increment between bins (delta) and the scheme to use for storing the data (input_fmt). The data in a vector data set are taken as a function of a variable M. If M is allowed to vary over the individual measurement period or if M actually represents a band width, then each element in the vector can be considered to have been accumulated with the interval M - δ1 to M + δ2. Vector data is binned (along the rows) by M. If the user selects the POINT STORAGE scheme, the data is stored by the center variable M. If the center variable M is located between the upper and lower edge values of a given bin, the data value is placed only in this bin. If the user selects the BAND STORAGE scheme, the data is placed in all bins which
are fully or partially contained within the range $M - \delta_1$ to $M + \delta_2$. The data is multiplied by the percentage of the bin covered by the range before the data is placed into the bin.

The bin_fill parameter defines the method that is to be used to fill in bins that have not been filled in with data. If the data bins are to be left as is, with the unfilled bins left unfilled, the user should set this parameter to NO_BIN_FILL. When selecting a fill method, the user must be aware that for some of the virtual instruments, the binning of the data occurs within a two-dimensional set of bins. In this 2-D binning matrix, the columns represent the data bins and the rows represent phi or azimuthal bins. If the sensor measurements are independent of phi, the binning of the data is only one-dimensional; otherwise, the binning is two-dimensional. In both 1-D and 2-D binning, missing or unfilled bins can be filled by linearly interpolating across the holes using values defined at adjacent bins (LINEAR) or the data in the adjacent bins can be projected inward across the area of missing bins meeting in the center of the gap (CONSTANT). For 2-D binning, such filling can either occur first along the columns and then along the rows (COL/ROW), or first along the rows and then along the columns (ROW/COL). The 2-D LEAST SQUARES FIT fill method is selectable only for the 2-D data binning. If the user selects this fill method for 1-D data binning, the set_bin_info module will change the option to LINEAR COL/ROW (LIN_COL_ROW).

The user should be aware that the data buffers that come back from the time-averaged, sample-averaged, and spin-averaged routines are NOT modified as far as missing bins is concerned. If the user wishes to fill in the missing bins according to the bin_fill parameter, the user must call the module buffer_bin_fill. If the data are collapsed over specified dimensions, the buffer_bin_fill module need not be called.

Some of the error codes returned by this module are the error codes returned by the module set_collapse_info. With regards to calling sequences, if the set_bin_info module is to be utilized, it should be called before the set_collapse_info module is called since the set_collapse_info module allocates space based upon the number of bins for the data buffers. If it is determined that the set_collapse_info module was called prior to calling the set_bin_info routine, the set_collapse_info routine will be recalled from within the set_bin_info module in order to allocate space for the correct number of bins.

The parameter swp_fmt specifies how the band width values are to be calculated using the center sweep step values. Two of the options, zero spacing (ZERO_SPACING) and variable width spacing (VARIABLE_SPACING) are applicable only for FIXED_SWEEP processing. If the user tries to specify these values for VARIABLE_SWEEP processing, an error code is returned. Zero spacing defines a scheme where the lower edge of the band is the same as the upper edge of the band; that is, the band width values are the same as the center values. Linear spacing defines a scheme where the lower (upper) edge of the band is determined by subtracting (adding) one-half of the difference between two successive center values from (to) the center value. The same algorithm is used for log spacing, with the log of the center values being utilized. Variable width spacing defines a scheme which makes use of tables defined in the VIDF file to create the center and band width values. The parameters num_center, center_tbls and center_opers define the tables and table operations that are to be utilized to calculate the center sweep step values. The parameters
**num_band**, **band_tbls** and **band_opers** define the tables and table operations that are to be utilized to calculate correction values that are to be applied to the center values in order to calculate the band width values. The variable **var_fmt** specifies how the correction values are to be applied. If the **var_fmt** value is 'L' or 'l', the lower edge of the band is set to the center value and the upper edge of the band is calculated by adding the correction value to the center value. If the **var_fmt** value is 'C' or 'c', the lower edge of the band is calculated by subtracting one-half of the correction value from the center value and the upper edge of the band is calculated by adding one-half of the correction value to the center value. If the **var_fmt** value is 'U' or 'u', the lower edge of the band is calculated by subtracting the correction value from the center value and the upper edge of the band is set to the center value. If the **var_fmt** value is 'E' or 'e', the lower edge of the band is set to the center value and the upper edge of the band is set to the correction value; therefore, the correction value is not really a delta value, it is the actual value to be used as the upper edge of the band. If this format is selected, please take note that the center values and the lower edge values will be identical. If the **var_fmt** value is 'A' or 'a', there is no need to perform a computation using the center values in order to derive the lower and upper edges of the band. The “actual” values for the centers, lower edges and upper edges of the scan band are defined within the VIDF. The parameters **num_center**, **center_tbls**, and **center_opers** define the tables and table operations that are to be utilized to calculate the center sweep step values. The parameters **num_band**, **band_tbls**, and **band_opers** define the tables and table operations that are to be utilized to calculate the lower edges of the band. The parameters **num_upper_band**, **upper_band_tbls**, and **upper_band_opers** define the tables and table operations that are to be utilized to calculate the upper edges of the band. The user is referred to the **center_and_band_values** write-up for more information concerning center and band width values.

**ERRORS**

All errors within this routine are returned through the status variable. The include file **ret_codes.h**, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The **ret_codes.h** file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- file_open 1R
- fill_data 2R
- fill_discontinuous_data 2R
- fill_mode_data 2R
- sweep_data 2R
- sweep_discontinuous_data 2R
- sweep_mode_data 2R
- spin_data 2R
- spin_data_pixel 2R
- set_scan_info 2R
- buffer_bin_fill 2R
- get_data_key 1R
- setCollapse_info 2R
EXAMPLES
Create the data bins using the FIXED SWEEP/LINEAR SPACED binning scheme and leave empty bins unprocessed for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"
#define DUMMY_VAL 0

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_LONG *no_opers = NULL;
SDDAS_SHORT status;
SDDAS_CHAR *no_tbls = NULL;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = set_bin_info (data_key, ",", vnum, FIXED_SWEEP, DUMMY_VAL, DUMMY_VAL, DUMMY_VAL, DUMMY_VAL, DUMMY_VAL, DUMMY_VAL, LINE_SPACING, 0, no_tbls, no_opers, 0, no_tbls, no_opers, 0, no_tbls, no_opers, '\0', DUMMY_VAL, NO_BIN_FILL);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by set_bin_info routine.\n", status);
    exit (-1);
}
```
Create sixteen bins, starting at 5ev, stopping at 155ev, using a delta of 10ev per bin, with log spacing and the data is to be stored using BAND STORAGE. Empty bins are be left alone.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_LONG *no_opers = NULL;
SDDAS_SHORT status;
SDDAS_CHAR *no_tbls = NULL;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by get_data_key routine.\n", status);
        exit (-1);
    }
get_version_number (&vnum);

status = set_bin_info (data_key, ",", vnum, VARIABLE_SWEEP, 5.0, 155.0, 10.0, 16,
                      LOG_SPACING, 0, no_tbls, no_opers, 0, no_tbls,
                      no_opers, 0, no_tbls, no_opers, ",0", BAND_STORAGE,
                      NO_BIN_FILL);
if (status != ALL_OKAY)
    {
        printf ("\n Error %d returned by set_bin_info routine.\n", status);
        exit (-1);
    }

Create the data bins using the FIXED SWEEP/VARIABLE WIDTH SPACED binning scheme and fill in the empty bins using a constant row/column approach for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_SHORT status;
SDDAS_LONG center_opers[1], lower_band_opers[2], upper_band_opers[2];
SDDAS_CHAR center_tbls[1], lower_band_tbls[2], upper_band_tbls[2];
SDDAS_CHAR num_center, num_lower_band, num_upper_band;
```
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

num_center = 1;
center_tbls[0] = 5;
center_opers[0] = 0;
num_lower_band = 2;
lower_band_tbls[0] = 7;
lower_band_tbls[1] = 8;
lower_band_opers[0] = 0;
lower_band_opers[1] = 3;

num_upper_band = 2;
upper_band_tbls[0] = 9;
upper_band_tbls[1] = 10;
upper_band_opers[0] = 0;
upper_band_opers[1] = 3;

status = set_bin_info (data_key, ", vnum, FIXED_SWEEP, DUMMY_VAL,
    DUMMY_VAL, DUMMY_VAL, DUMMY_VAL,
    VARIABLE_SPACING, num_center, center_tbls, center_opers,
    num_lower_band, lower_band_tbls, lower_band_opers,
    num_upper_band, upper_band_tbls, upper_band_opers,
    'A', POINT_STORAGE, CON_ROW_COL);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by set_bin_info routine.\n", status);
    exit (-1);
}
**SET_COLLAPSE_INFO**

function - sets up information that is pertinent to the collapsing of the data over multiple dimensions

**SYNOPSIS**

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT set_collapse_info (SDDAS_Ulong data_key, SDDAS_CHAR *exten,
                           SDDAS_Ushort version, SDDAS_SHORT num_units,
                           SDDAS_FLOAT delta_phi, SDDAS_FLOAT *actual_phi,
                           SDDAS_CHAR interleave)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **num_units** - the total number of units or data levels to be returned for all sensors (must be the same for all sensors)
- **delta_phi** - the requested resolution of the phi bins
- **actual_phi** - the actual resolution used for the phi bins
- **interleave** - flag indicating if the data is to be interleaved (data not cleared out if missing on next sweep)
  - 0 - clear out data matrices and buffers upon each call to the fill_data / fill_discontinuous_data / sweep_data / sweep_discontinuous_data / spin_data / spin_data_pixel routine
  - 1 - leave the data matrices and buffers preserved (do not clear)

**set_collapse_info** - routine status (see TABLE 1)

**TABLE 1. Status Codes Returned for SET_COLLAPSE_INFO**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLAPSE_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>COLLAPSE_MALLOC</td>
<td>no memory to hold the data collapsing structures</td>
</tr>
<tr>
<td>COLLAPSE_SEN_MALLOC</td>
<td>no memory to hold sensor specific collapsing information</td>
</tr>
<tr>
<td>THETA_CHK_MALLOC</td>
<td>no memory for temporary array</td>
</tr>
<tr>
<td>THETA_BIN_MALLOC</td>
<td>no memory for theta angle values</td>
</tr>
<tr>
<td>ORDER_THETA_MALLOC</td>
<td>no memory for theta bin order indexes</td>
</tr>
<tr>
<td>COLLAPSE_DATA_MALLOC</td>
<td>no memory for matrices that hold data gathered over specific dimensions</td>
</tr>
<tr>
<td>COLLAPSE_DATA_ADDRESS</td>
<td>no memory to hold the array of addresses of the pointers to the data and normalization factors for the phi matrices</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>
DESCRIPTION

Set_collapse_info is the IDFS routine which sets up the data matrices and other information that is pertinent to collapse data over the charge, mass, phi, theta and/or scan dimensions. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The first call to the set_collapse_info module for the data set specified will be used to generate the dimension collapsing information. All subsequent calls specifying the same data set will be ignored. In addition, if the module set_bin_info is to be utilized, it should be called prior to calling this module since this module allocates space based upon the number of bins per data buffer. If the only type of data to be processed for the data set in question is instrument status (mode) data, the user does not need to call this module since dimensionality is associated with sensor-specific data and instrument status data is not sensor-specific.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

If the virtual instrument acquires data as a function of phi and the user wishes to average the data over a specified phi range, the fill_data / sweep_data / spin_data / spin_data_pixel routine must be used to acquire the phi data matrix. The user must call the set_collapse_info module before any data is gathered in order to specify the resolution of the phi bins and to specify if the interleave option is to be utilized when building the phi matrix. If the phi data matrix is not needed, the user should pass 360.0 for the delta_phi parameter. The fill_discontinuous_data / sweep_discontinuous_data routine does not support any data sets with a PHI dimension; therefore, the user should pass 360.0 for the delta_phi parameter if the fill_discontinuous_data / sweep_discontinuous_data routine is to be utilized. The parameter actual_phi returns the true resolution used for the phi bins. An internal calculation (360.0 / delta_phi) is performed to ensure that an integer number of phi bins results; therefore, in some cases, the resolution actually used may deviate from the
resolution specified. The interleave option will be utilized for all data matrices that are associated with a specific dimension.

The default mode for the fill_data / fill_discontinuous_data / sweep_data / sweep_discontinuous_data / spin_data / spin_data_pixel routines is to return sensor data in one data level (raw units) for each of the sensors processed. If the default mode is preserved, the value for the parameter num_units should be set to one. If the user made any calls to the fill_sensor_info module to add/modify the data level(s) being returned, the value for the parameter num_units can be retrieved by calling the module units_index.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
fill_data 2R
fill_discontinuous_data 2R
sweep_data 2R
sweep_discontinuous_data 2R
spin_data 2R
spin_data_pixel 2R
fill_sensor_info 2R
units_index 2R
set_bin_info 2R
get_data_key 1R
get_version_number 1R
ret_codes 1H
libtrec_idfs 2H

BUGS
None

EXAMPLES
In order to produce a line plot from the RTLA virtual instrument, data must be collapsed over a frequency range. Assuming that a single data level (raw units) is being returned, make a call to disable phi information and the interleave option. The virtual instrument RTLA is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

#include "libtrec_idfs.h"
#include "ret_codes.h"
SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT actual_phi;
SDDAS_SHORT status, num_units;

num_units = 1;
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = set_collapse_info (data_key, ",", vnum, num_units, 360.0, &actual_phi, 0);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by set_collapse_info routine.\n", status);
    exit (-1);
}
**SET_SCAN_INFO**

function - specifies the tables/operations to be applied to arrive at the desired units for the sweep step values that are associated with the data bins

**SYNOPSIS**

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT set_scan_info (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                       SDDAS_USHORT version, SDDAS_CHAR num_tbls,
                       SDDAS_CHAR *tbls_to_apply, SDDAS_LONG *tbl_oper)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **num_tbls** - the number of elements specified in the **tbls_to_apply** and **tbl_oper** parameters
- **tbls_to_apply** - the tables that are to be applied in order to derive the desired units for the sweep step values
- **tbl_oper** - the operations that are to be applied to the specified tables in order to derive the desired units for the sweep step values
- **set_scan_info** - routine status (see TABLE 1)

**TABLE 1. Status Codes Returned for SET_SCAN_INFO**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_SCAN_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>SCAN_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call set_bin_info for this combination)</td>
</tr>
<tr>
<td>SCAN_INDEX_MALLOC</td>
<td>no memory to hold table offset values</td>
</tr>
<tr>
<td>SET_SCAN_TBL_MALLOC</td>
<td>no memory for table number / table operation information</td>
</tr>
<tr>
<td>SCAN_IDF_ELE_NOT_FOUND</td>
<td>the data item being requested was not found in the VIDF file</td>
</tr>
<tr>
<td>SCAN_IDF_MANY_BYTES</td>
<td>the number of elements being requested is more than the number of elements available for the selected field</td>
</tr>
<tr>
<td>SCAN_IDF_TBL_NUM</td>
<td>the table being requested exceeds the number of defined tables</td>
</tr>
<tr>
<td>SCAN_IDF_CON_NUM</td>
<td>the constant being requested exceeds the number of defined constants</td>
</tr>
<tr>
<td>SCAN_IDF_NO_ENTRY</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

Set_scan_info is the IDFS routine that is utilized to specify the units for the sweep step values that are associated with the data bins defined by the set_bin_info module. These sweep step values are used by the IDFS routines that return time-averaged data (fill_data / fill_discontinuous_data), sample-averaged data (sweep_data /...
sweep_discontinuous_data), or spin-averaged data (spin_data / spin_data_pixel) when storing the data into the data bins. This module must be called after the call to the set_bin_info module has been made; otherwise, an error code is returned. A call to this routine is optional since the set_bin_info module sets up the system to calculate the sweep step values in terms of raw units, with one set of sweep step values defined for all sensors for the selected data source. If the user intends to make use of this module, the call must be made prior to calling the center_and_band_values routine. If the only type of data to be processed for the data set in question is instrument status (mode) data, the user does not need to call this module since the fill_mode_data / sweep_mode_data routine determines how the data is to be stored in the data bins.

The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. The first call to the set_scan_info module which indicates non-raw units will be used to specify the units for the sweep step values. All subsequent calls specifying the same data set will be ignored. This routine should be used with FIXED_SWEEP, non-variable width spaced data bins. If the user selected VARIABLE_SWEEP or FIXED_SWEEP with data bins that are created using VARIABLE_SPACING, the information specified by this routine is stored but not utilized when determining the sweep step values that are associated with the data bins; the information specified in the call to the set_bin_info routine is utilized.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

The units for the sweep step values associated with the data bins is specified by the user through the parameters num_tbls, tbls_to_apply and tbl_oper. If the user wants raw units, that is, the telemetry data, to be returned, the user should set the num_tbls parameter to zero and put a placeholder variable for the tbls_to_apply and tbl_oper parameters. For
other units, the user must specify the tables and the table operations that are to be applied to calculate the desired unit. The order is implied by the contents of the \texttt{tbsls\_to\_apply} array.

**ERRORS**

All errors within this routine are returned through the status variable. The include file \texttt{ret\_codes.h}, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The \texttt{ret\_codes.h} file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- \texttt{fill\_data} 2R
- \texttt{fill\_discontinuous\_data} 2R
- \texttt{fill\_mode\_data} 2R
- \texttt{sweep\_data} 2R
- \texttt{sweep\_discontinuous\_data} 2R
- \texttt{sweep\_mode\_data} 2R
- \texttt{spin\_data} 2R
- \texttt{spin\_data\_pixel} 2R
- \texttt{set\_bin\_info} 2R
- \texttt{center\_and\_band\_values} 2R
- \texttt{file\_open} 1R
- \texttt{get\_data\_key} 1R
- \texttt{get\_version\_number} 1R
- \texttt{ret\_codes} 1H
- \texttt{libtrec\_idfs} 2H

**BUGS**

None

**EXAMPLES**

Indicate the tables that are to be applied to derive the sweep step values for the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec\_idfs.h"
#include "ret\_codes.h"

SDDAS\_ULONG data\_key;
SDDAS\_USHORT vnum;
SDDAS\_LONG tbl\_oper[2];
SDDAS\_SHORT status;
SDDAS\_CHAR num\_tbs, tbs\_to\_apply[2];
```
status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

/* There are two tables to be applied to derive sweep step units.    */
/* Table 3 is the first table to be applied, followed by table 5.    */

num_tbls = 2;
tbls_to_apply[0] = 3;
tbls_to_apply[1] = 5;
tbl_oper[0] = 0;
tbl_oper[1] = 3;

status = set_scan_info (data_key, "", vnum, num_tbls, tbls_to_apply, tbl_oper);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by set_scan_info routine.\n", status);
    exit (-1);
}
**SET_TIME_VALUES**

Function - sets the base reference time, reference location and time duration to be utilized by the fill_data / fill_discontinuous_data / fill_mode_data / spin_data_pixel modules.

**SYNOPSIS**

```c
#include "libtrec_idfs.h"

void set_time_values (SDDAS_USHORT version, SDDAS_SHORT base_year, SDDAS_SHORT base_day, SDDAS_LONG base_sec, SDDAS_LONG base_nano, SDDAS_LONG base_pix, SDDAS_LONG res_sec, SDDAS_LONG res_nano);
```

**ARGUMENTS**

- **version** - IDFS data set identification number which allows for multiple openings of the same data set.
- **base_year** - the year time component for the base reference time.
- **base_day** - the day time component for the base reference time.
- **base_sec** - the time of day in seconds for the base reference time.
- **base_nano** - the time of day residual in nanoseconds for the base reference time.
- **base_pix** - the reference point or location associated with the base reference time.
- **res_sec** - the time duration (delta) in seconds.
- **res_nano** - the time duration residual in nanoseconds.

**DESCRIPTION**

`set_time_values` is the IDFS routine that sets the base reference time, the reference location and the time duration values to be used by the fill_data / fill_discontinuous_data / fill_mode_data / spin_data_pixel routines. This routine should be called once, after all calls to the file_open and file_pos routines have been made. If the base reference time is not known, as is the case in real-time processing, the user can make a call to the read_drec / read_drec_spin module in order to retrieve the starting time of the first sweep processed.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the **get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

**ERRORS**

This routine returns no status or error codes.
SEE ALSO

- fill_data 2R
- fill_discontinuous_data 2R
- fill_mode_data 2R
- spin_data_pixel 2R
- file_open 1R
- file_pos 1R
- get_version_number 1R
- read_drec 1R
- read_drec_spin 1R
- libtrec_idfs 2H

BUGS

None

EXAMPLES

The base reference time to be utilized is 1992, day 23, time 00:25:36 which is equal to 1536 seconds. The resolution to be utilized is 1.500 seconds and the reference location is at zero. Assume that the variable vnum has been set by a previous call to the get_version_number routine.

```
set_time_values (vnum, 1992, 23, 1536, 0, 0, 1, 500000000);
```
SPIN_DATA
function - returns spin-averaged sensor-specific data

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT spin_data (SDDAS ULONG data_key, SDDAS CHAR *exten,
SDDAS USHORT version, void *idf_data_ptr,
SDDAS SHORT **ret_sensors, SDDAS FLOAT **ret_data,
SDDAS FLOAT **ret_frac, SDDAS CHAR **bin_stat,
SDDAS SHORT *num_sen, SDDAS SHORT **num_units,
SDDAS SHORT *block_size, SDDAS SHORT **stime_yr,
SDDAS SHORT **stime_day, SDDAS LONG **stime_sec,
SDDAS_LONG **stime_nano, SDDAS_SHORT **etime_yr,
SDDAS_SHORT **etime_day, SDDAS_LONG **etime_sec,
SDDAS_LONG **etime_nano, SDDAS CHAR *hdr_change)

ARGUMENTS
data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple
openings of the same data set
idf_data_ptr - pointer to the idf_data structure that temporarily holds
sensor data and pertinent ancillary data for the data set of
interest
ret_sensors - an array which holds the sensor number(s) for which data is
returned
   - the array is initialized to -1 in all elements;
   - valid sensor numbers start with 0
ret_data - pointer to the data being returned (data for all sensors
processed)
ret_frac - pointer to the normalization factors for the data being returned
bin_stat - pointer to status flags which are associated with each data bin
   returned
   0 - no data has been placed into the data
   bin being processed
   1 - data has been placed into the data bin
   being processed
num_sen - the number of elements in the ret_sensors array
num_units - an array holding the number of data sets to bypass in order to
get to the data for the sensor being processed
block_size - the number of data values returned in a data buffer
stime_yr - pointer to the start time year value for the first sweep
contained within the spin being processed
spin_data (2R)  spin_data (2R)

stime_day - pointer to the start time day of year value for the first sweep contained within the spin being processed
stime_sec - pointer to the start time of day value (in seconds) for the first sweep contained within the spin being processed
stime_nano - pointer to the start time of day residual (in nanoseconds) for the first sweep contained within the spin being processed
etime_yr - pointer to the end time year value for the last sweep contained within the spin being processed
etime_day - pointer to the end time day of year value for the last sweep contained within the spin being processed
etime_sec - pointer to the end time of day value (in seconds) for the last sweep contained within the spin being processed
etime_nano - pointer to the end time of day residual (in nanoseconds) for the last sweep contained within the spin being processed
hdr_change - flag which indicates a header change occurred while processing the data

0 - a header change was not encountered during the processing of the data
1 - a header change was encountered during the processing of the data

spin_data - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SPIN_DATA

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIN_DATA_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>SPIN_DATA_NO_SPIN</td>
<td>the requested data set does not spin</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which holds information pertinent to the spin-averaged data</td>
</tr>
<tr>
<td>SPIN_DATA_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call set_bin_info for this combination)</td>
</tr>
<tr>
<td>SPIN_DATA_CENTER_BAND_MISSING</td>
<td>the routine center_and_band_values has not been called prior to calling the spin_data routine</td>
</tr>
<tr>
<td>SPIN_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SPIN_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SPIN_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SPIN_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>SPIN_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>SPIN_DATA_MALLOC</td>
<td>no memory for data buffer</td>
</tr>
<tr>
<td>SPIN_DATA_WITH_FILL_SWEEP</td>
<td>spin_data cannot be used interchangeably with the fill_data and / or sweep_data routines for the same data key, extension, version combination</td>
</tr>
<tr>
<td>PHI_DIFF_UNITS</td>
<td>the sensors being processed do not process the same number of data levels (units)</td>
</tr>
<tr>
<td>FILL_PHI_FIRST</td>
<td>the starting azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
</tbody>
</table>
### DESCRIPTION

**Spin_data** is the IDFS spin-averaging data read routine for sensor-specific data, summing the data for all sweeps that pertain to the spin being processed. The data set of interest is referenced through the key value **data_key** which can be created using the **get_data_key** module. **Spin_data** processes sensor-specific data only, that is, it can process sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data; however, the type of manipulation being performed by this module is best suited to sensor data. **Spin_data** assumes that the data set of interest does not roll over to a minimum value when the maximum threshold has been reached or to a maximum value when the minimum threshold has been reached. This assumption is crucial since multiple samples are averaged together. If the data set does roll over at the thresholds, the averaging of these samples will probably result in incorrect data values. An example of a roll over data set is longitude data, which resets values to the minimum threshold (-180) when the maximum threshold (180) has been reached.

For the **spin_data** module, data acquisition is based upon a full spin of data, not a given time interval as with the **fill_data** module; however, there is still the need to utilize some form of time control. The sensor that is selected as the controller through the call to the **start_of_spin** module will serve as the time manager and all sensors that have a start of spin time WITHIN the time interval of the controller's spin will be returned as a group.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the **get_version_number** routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the **file_open** routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable **exten**. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA,
which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

This routine will process data one spin at a time, placing the data into one buffer which holds data that is accumulated over the current spin interval. There are N many sub-buffers which hold the data in each of the requested data levels or units. The user must process the data contained within the data buffer before the next call to the spin_data routine is made since the module will clear out this buffer for re-use. The data values must be normalized using the normalization factors returned along with the data. If the sensors rotate or alternate when data is returned, the result may be that the data buffer for a specific sensor may not contain any data since the data buffer is reset or cleared out upon each call to the spin_data module. The user is advised to check the value or values in the bin_stat array. If all values are 0, no data was placed into the buffer.

The size and spacing of the data buffers are either defined by the user or by elements contained within the virtual instrument definition document. The user must call the set_bin_info module before calling the spin_data routine in order to specify how the binning of the data is to occur. In addition, the user must call the center_and_band_values module before calling the spin_data module. If the spin_data routine determines that no binning scheme has been selected, an error code is returned to the user.

The user should be aware that the data buffers that come back from the spin_data module are NOT modified as far as missing bins are concerned. If the user wishes to fill in the missing bins according to the method specified in the call to the set_bin_info module, the user must call the module buffer_bin_fill. If the data are collapsed over specified dimensions, the buffer_bin_fill module need not be called.

The default mode for the spin_data routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the fill_sensor_info module prior to calling the spin_data module. The user should make one call to the fill_sensor_info module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

If the virtual instrument acquires data over the PHI dimension and the user wishes to average the data over a specified phi range, the spin_data routine must be used to acquire the phi data matrix. The user must call the module set-collapse_info prior to calling the spin_data routine in order to specify the resolution of the phi bins and to specify if the interleave option is to be utilized when building the phi matrix.

ERRORS

All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the
mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO

file_open  1R
file_pos  1R
start_of_spin  1R
get_data_key  1R
get_version_number  1R
create_data_structure  1R
create_idf_data_structure  1R
set_bin_info  2R
center_and_band_values  2R
set_scan_info  2R
fill_sensor_info  2R
units_index  2R
setCollapse_info  2R
buffer_bin_fill  2R
ret_codes  1H
user_defs  1H
libtrec_idfs  2H

BUGS

None

EXAMPLES

Obtain spin-averaged data from the virtual instrument CP3DRH, which is part of the 3DR instrument, which is part of the PEACE experiment, which is part of the CLUSTER-2 mission, which is identified with the CLUSTERII project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
SDDAS_SHORT *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change;
void *idf_data_ptr;

status = get_data_key ("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH",
                     &data_key);
```
if (status != ALL_OKAY)
{
    printf ("% Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);
.
.
.

status = spin_data (data_key, ",", vnum, idf_data_ptr, &sen_numbers,
    &ret_data, &ret_frac, &ret_bin, &num_sen, &num_units,
    &data_block, &start_time_yr, &start_time_day,
    &start_time_sec, &end_time_yr, &end_time_day, &end_time_sec,
    &hdr_change);
if (status != ALL_OKAY)
{
    printf("n Error %d returned by spin_data routine.\n", status);
    exit (-1);
}
**SPIN_DATA_PIXEL**

function - returns spin-averaged sensor-specific data along with reference indicators with respect to a time-oriented axis

**SYNOPSIS**

```
#include "libtrec_idfs.h"
#include "ret_codes.h"
```

```
SDDAS_SHORT spin_data_pixel (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
                          SDDAS_USHORT version, void *idf_data_ptr,
                          SDDAS_SHORT **ret_sensors, SDDAS_FLOAT **ret_data,
                          SDDAS_FLOAT **ret_frac, SDDAS_CHAR **bin_stat,
                          SDDAS_LONG **bpix, SDDAS_LONG **epix,
                          SDDAS_SHORT *num_sen, SDDAS_SHORT **num_units,
                          SDDAS_SHORT *block_size, SDDAS_SHORT **stime_yr,
                          SDDAS_SHORT **stime_day, SDDAS_LONG **stime_sec,
                          SDDAS_LONG **stime_nano, SDDAS_SHORT **etime_yr,
                          SDDAS_SHORT **etime_day, SDDAS_LONG **etime_sec,
                          SDDAS_LONG **etime_nano, SDDAS_CHAR *hdr_change)
```

**ARGUMENTS**

- **data_key** - unique value which indicates the data set of interest
- **exten** - two character extension to be added to IDFS file names when default files are not to be used, otherwise a null string
- **version** - IDFS data set identification number which allows for multiple openings of the same data set
- **idf_data_ptr** - pointer to the *idf_data* structure that temporarily holds sensor data and pertinent ancillary data for the data set of interest
- **ret_sensors** - an array which holds the sensor number(s) for which data is returned
  - the array is initialized to -1 in all elements; valid sensor numbers start with 0
- **ret_data** - pointer to the data being returned (data for all sensors processed)
- **ret_frac** - pointer to the normalization factors for the data being returned
- **bin_stat** - pointer to status flags which are associated with each data bin returned
  - 0 - no data has been placed into the data bin being processed
  - 1 - data has been placed into the data bin being processed
- **bpix** - pointer to the starting pixel location for the data buffer returned
- **epix** - pointer to the ending pixel location for the data buffer returned
num_sen - the number of elements in the ret_sensors array
num_units - an array holding the number of data sets to bypass in order to get to the data for the sensor being processed
block_size - the number of data values returned in a data buffer
stime_yr - pointer to the start time year value for the first sweep contained within the spin being processed
stime_day - pointer to the start time day of year value for the first sweep contained within the spin being processed
stime_sec - pointer to the start time of day value (in seconds) for the first sweep contained within the spin being processed
stime_nano - pointer to the start time of day residual (in nanoseconds) for the first sweep contained within the spin being processed
etime_yr - pointer to the end time year value for the last sweep contained within the spin being processed
etime_day - pointer to the end time day of year value for the last sweep contained within the spin being processed
etime_sec - pointer to the end time of day value (in seconds) for the last sweep contained within the spin being processed
etime_nano - pointer to the end time of day residual (in nanoseconds) for the last sweep contained within the spin being processed
hdr_change - flag which indicates a header change occurred while processing the data
    0 - a header change was not encountered during the processing of the data
    1 - a header change was encountered during the processing of the data
spin_data_pixel - routine status (see TABLE 1)

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIN_DATA_PIX_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>SPIN_DATA_PIX_NO_SPIN</td>
<td>the requested data set does not spin</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which holds information pertinent to the spin-averaged data</td>
</tr>
<tr>
<td>SPIN_DATA_PIX_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call set_bin_info for this combination)</td>
</tr>
<tr>
<td>SPIN_DATA_PIX_CENTER_BAND_MISSING</td>
<td>the routine center_and_band_values has not been called prior to calling the spin_data_pix routine</td>
</tr>
<tr>
<td>SPIN_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SPIN_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SPIN_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SPIN_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>SPIN_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>SPIN_DATA_MALLOC</td>
<td>no memory for data buffer</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>SPIN_DATA_PIX_WITH_FILL_SWEEP</td>
<td>spin_data_pix cannot be used interchangeably with the fill_data and / or sweep_data routines for the same data key, extension, version combination</td>
</tr>
<tr>
<td>BAD_VFMT</td>
<td>bad format character for variable width bin spacing</td>
</tr>
<tr>
<td>SWP_TIMES_TMP_MALLOC</td>
<td>no memory to hold the time components for each element of the sweep</td>
</tr>
<tr>
<td>PHI_DIFF_UNITS</td>
<td>the sensors being processed do not process the same number of data levels (units)</td>
</tr>
<tr>
<td>FILL_PHI_FIRST</td>
<td>the starting azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td>FILL_PHI_LAST</td>
<td>the ending azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

Spin_data_pix is the IDFS spin-averaging data read routine for sensor-specific data, summing the data for all sweeps that pertain to the spin being processed. The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. Spin_data_pix processes sensor-specific data only, that is, it processes sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data; however, the type of manipulation being performed by this module is best suited to sensor data. Spin_data_pix assumes that the data set of interest does not roll over to a minimum value when the maximum threshold has been reached or to a maximum value when the minimum threshold has been reached. This assumption is crucial since multiple samples may be averaged together in a single buffer. If the data set does roll over at the thresholds, the averaging of these samples will probably result in incorrect data values. An example of a roll over data set is longitude data, which resets values to the minimum threshold (-180) when the maximum threshold (180) has been reached.

For the spin_data_pix module, data acquisition is based upon a full spin of data, not a given time interval as with the fill_data module; however, there is still the need to utilize some form of time control. The sensor that is selected as the controller through the call to the start_of_spin module will serve as the time manager and all sensors that have a start of spin time WITHIN the time interval of the controller's spin will be returned as a group. The difference between the spin_data module and this module is the return of a starting location and an ending location that are located along a time-axis, similar to the fill_data routine. The user may use these values as references to the base location specified in the call to the set_time_values module. That is, given a base time value, a time interval and a reference location, the spin_data_pix routine will return the location of the spin with respect to time. The user may chose to ignore these values or may use these locations to plot data along an axis that is scaled with respect to time. The user **must** call the module set_time_values before the spin_data_pix module can be called. If the spin_data_pix routine determines that the set_time_values module has not been called, an error code is returned to the user.
The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTL19922181432Dxx, RTL19922181432Hxx, RTL19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

This routine will process data one spin at a time, placing the data into a buffer which holds data that is accumulated over the current spin interval. There are N many sub-buffers which hold the data in each of the requested data levels or units. The user must process the data contained within the data buffer before the next call to the spin_data_pixel routine is made since the module will clear out this buffer for re-use. The data values must be normalized using the normalization factors returned along with the data. If the sensors rotate or alternate when data is returned, the result may be that the data buffer for a specific sensor may not contain any data since the data buffer is reset or cleared out upon each call to the spin_data_pixel module. The user is advised to check the value or values in the bin_stat array. If all values are 0, no data was placed into the buffer.

The size and spacing of the data buffer is either defined by the user or by elements contained within the virtual instrument definition document. The user must call the set_bin_info module before calling the spin_data_pixel routine in order to specify how the binning of the data is to occur. In addition, the user must call the center_and_band_values module before calling the spin_data_pixel module. If the spin_data_pixel routine determines that no binning scheme has been selected, an error code is returned to the user.

The user should be aware that the data buffer that comes back from the spin_data_pixel routine is NOT modified as far as missing bins is concerned. If the user wishes to fill in the missing bins according to the method specified in the call to the set_bin_info module, the user must call the module buffer_bin_fill. If the data are collapsed over specified dimensions, the buffer_bin_fill module need not be called.
The default mode for the spin_data_pix routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the fill_sensor_info module prior to calling the spin_data_pix module. The user should make one call to the fill_sensor_info module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

If the virtual instrument acquires data over the PHI dimension and the user wishes to average the data over a specified phi range, the spin_data_pix routine must be used to acquire the phi data matrix. The user must call the module set_collapse_info prior to calling the spin_data_pix module in order to specify the resolution of the phi bins and to specify if the interleave option is to be utilized when building the phi matrix.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
file_pos 1R
start_of_spin 1R
get_data_key 1R
get_version_number 1R
create_data_structure 1R
create_idf_data_structure 1R
set_bin_info 2R
center_and_band_values 2R
set_scan_info 2R
fill_sensor_info 2R
units_index 2R
setCollapse_info 2R
buffer_bin_fill 2R
set_time_values 2R
ret_codes 1H
user_defs 1H
libtrec_idfs 2H

BUGS
None
EXAMPLES
Obtain spin-averaged data from the virtual instrument CP3DRH, which is part of the 3DR instrument, which is part of the PEACE experiment, which is part of the CLUSTER-2 mission, which is identified with the CLUSTERII project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_Ulong data_key;
SDDAS_Ushort vnum;
SDDAS_Float *ret_data, *ret_frac;
SDDAS_Long *start_time_sec, *start_time_nano, *end_time_sec, *end_time_nano;
SDDAS_Long *bpix, *epix;
SDDAS_Short *start_time_yr, *start_time_day, *end_time_yr, *end_time_day;
SDDAS_Short status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_Char *ret_bin, hdr_change;
void *idf_data_ptr;

status = get_data_key ("CLUSTERII", "CLUSTER-2", "PEACE", "3DR", "CP3DRH",
                       &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = spin_data_pix (data_key, ",", vnum, idf_data_ptr, &sen_numbers, &ret_data,
                        &ret_frac, &ret_bin, &bpix, &epix, &num_sen, &num_units,
                        &data_block, &start_time_yr, &start_time_day, &start_time_sec,
                        &start_time_nano, &end_time_yr, &end_time_day,
                        &end_time_sec, &end_time_nano, &hdr_change);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by spin_data_pix routine.\n", status);
    exit (-1);
}
```
Sweep_data function - returns sample-averaged sensor-specific data

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT sweep_data (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, void *idf_data_ptr,
SDDAS_LONG num_swps, SDDAS_SHORT **ret_sensors,
SDDAS_FLOAT **ret_data, SDDAS_FLOAT **ret_frac,
SDDAS_CHAR **bin_stat, SDDAS_SHORT *num_sen,
SDDAS_SHORT **num_units, SDDAS_SHORT *block_size,
SDDAS_SHORT *stime_yr, SDDAS_SHORT *stime_day,
SDDAS_LONG *stime_sec, SDDAS_LONG *stime_nano,
SDDAS_SHORT *etime_yr, SDDAS_SHORT *etime_day,
SDDAS_LONG *etime_sec, SDDAS_LONG *etime_nano,
SDDAS_CHAR *hdr_change, SDDAS_UCHAR exclude_dqual,
SDDAS_CHAR *complete_acq)

ARGUMENTS

- data_key - unique value which indicates the data set of interest
- exten - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
- version - IDFS data set identification number which allows for multiple
openings of the same data set
- idf_data_ptr - pointer to the idf_data structure that temporarily holds sensor
data and pertinent ancillary data for the data set of interest for
a single sweep of data
- num_swps - the number of samples (sweeps) to average together
- ret_sensors - an array which holds the sensor number(s) for which data is returned
- num_sen - the number of elements in the ret_sensors array
- num_units - an array holding the number of data sets to bypass in order to
get to the data for the sensor being processed
block_size  -  the number of data values returned in a data buffer  
stime_yr   -  the year value for the first sweep processed  
stime_day  -  the day of year value for the first sweep processed  
stime_sec  -  the time of day in seconds for the first sweep processed  
stime_nano -  the time of day residual in nanoseconds for the first sweep processed  
etime_yr   -  the year value for the last sweep processed  
etime_day  -  the day of year value for the last sweep processed  
etime_sec  -  the time of day in seconds for the last sweep processed  
etime_nano -  the time of day residual in nanoseconds for the last sweep processed  

hdr_change -  flag which indicates a header change occurred while processing the data  
  0 -  a header change was not encountered during the processing of the data  
  1 -  a header change was encountered during the processing of the data  

exclude_dqual -  data is to be excluded if the d_qual flag associated with the data is set to the value specified  

complete_acq -  flag which indicates if all samples (sweeps) were acquired  
  0 -  not all samples were acquired  
  1 -  all samples were acquired  

sweep_data  -  routine status (see TABLE 1)  

**TABLE 1.** Status Codes Returned for SWEEP_DATA

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWEEP_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which holds information pertinent to the sample-averaged data</td>
</tr>
<tr>
<td>SWEEP_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call set_bin_info for this combination)</td>
</tr>
<tr>
<td>SWEEP_CENTER_BAND_MISSING</td>
<td>the routine center_and_band_values has not been called prior to calling the sweep_data routine</td>
</tr>
<tr>
<td>SWEEP_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SWEEP_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SWEEP_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SWEEP_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>SWEEP_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>SWEEP_DATA_MALLOC</td>
<td>no memory for data buffer</td>
</tr>
<tr>
<td>SWEEP_WITH_FILL</td>
<td>the modules sweep_data and fill_data cannot be used interchangeably for the same data key, extension, version combination</td>
</tr>
<tr>
<td>BAD_VFMT</td>
<td>bad format character for variable width bin spacing</td>
</tr>
<tr>
<td>PHI_DIFF_UNITS</td>
<td>the sensors being processed do not process the same number of data levels (units)</td>
</tr>
<tr>
<td>FILL_PHI_FIRST</td>
<td>the starting azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
</tbody>
</table>
**DESCRIPTION**

*Sweep_data* is the IDFS sample-averaging data read routine for sensor-specific data, averaging *num_swps* sample sets (sweeps). The data set of interest is referenced through the key value *data_key* which can be created using the *get_data_key* module. *Sweep_data* processes sensor-specific data only, that is, it processes sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. If the instrument status (mode) data is desired, the user should use the *sweep_mode_data* routine. *Sweep_data* assumes that the data set of interest does not roll over to a minimum value when the maximum threshold has been reached or to a maximum value when the minimum threshold has been reached. This assumption is crucial since multiple samples may be averaged together. If the data set does roll over at the thresholds, the averaging of these samples will probably result in incorrect data values. An example of a roll over data set is longitude data, which resets values to the minimum threshold (-180) when the maximum threshold (180) has been reached. If the data set does roll over, the user should use the *sweep_discontinuous_data* routine. If the data set of interest is a combination of roll over and non-roll over data, for example, longitude data being returned along with science data, the user may use the *sweep_discontinuous_data* module in conjunction with the *sweep_data* routine, using the *sweep_data* routine to return the non-roll over data values and using the *sweep_discontinuous_data* routine to return the roll over data values. In order to do this correctly, the user must make use of multiple version numbers so that the same data files can be opened more than once. That is, use one version number for the non-roll over data and another version number for the roll over data. All IDFS routines that utilize a version number must be called once for each unique version number.

The data is processed one sweep at a time. Once the requested number of sweeps have been processed, the routine will return the data. If the requested number of sweeps could not be processed due to data acquisition problems (LOS_STATUS, NEXT_FILE_STATUS, EOF_STATUS), the routine will return the data and the normalization factors will reflect the number of sweeps processed so far. If more data is put online, the next call to the *sweep_data* routine will continue to accumulate data and will continue until the remaining sweeps have been acquired.

The parameter *version* allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the *get_version_number* routine to retrieve a unique version number.

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL_PHI_LAST</td>
<td>the ending azimuthal angle was not contained within any of the defined phi bins</td>
</tr>
<tr>
<td></td>
<td>error codes returned by <em>read_drec()</em></td>
</tr>
<tr>
<td></td>
<td>error codes returned by <em>convert_to_units()</em></td>
</tr>
<tr>
<td></td>
<td>error codes returned by <em>fill_sensor_info()</em></td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

### STATUS CODES

- **FILL_PHI_LAST**: The ending azimuthal angle was not contained within any of the defined phi bins. This occurs when an error code is returned by `read_drec()`, `convert_to_units()`, or `fill_sensor_info()`.

- **ALL_OKAY**: Routine terminated successfully.
instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter `idf_data_ptr` is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the `create_idf_data_structure` routine is made. The user also has the option of calling the module `create_data_structure`, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the `create_idf_data_structure` routine N times to create N instances of the `idf_data` structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

There are N many sub-buffers which hold the data in each of the requested data levels or units for each sensor. The user must process the data contained within these buffers before the next call to the `sweep_data` routine is made since the module will clear out these buffers for re-use if the requested number of sweeps were processed on the previous call. The data values must be normalized using the normalization factors returned along with the data. The user is advised to check the value or values in the `bin_stat` array. If all values are 0, no data was placed into the buffer. This can happen if the sensors rotate or alternate when data is returned or if the data is excluded based upon `d_qual` or data cutoff values.

The size and spacing of the data buffer is either defined by the user or by elements contained within the virtual instrument definition document. The user must call the `set_bin_info` module before calling the `sweep_data` routine in order to specify how the binning of the data is to occur. In addition, the user must call the `center_and_band_values` module before calling the `sweep_data` module. If the `sweep_data` routine determines that no binning scheme has been selected, an error code is returned to the user.

The user should be aware that the data buffer that comes back from the `sweep_data` routine are NOT modified as far as missing bins is concerned. If the user wishes to fill in the missing bins according to the method specified in the call to the `set_bin_info` module, the user must call the module `buffer_bin_fill`. If the data are collapsed over specified dimensions, the `buffer_bin_fill` module need not be called.
The default mode for the `sweep_data` routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the `fill_sensor_info` module prior to calling the `sweep_data` module. The user should make one call to the `fill_sensor_info` module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

If the virtual instrument acquires data over the PHI dimension and the user wishes to average the data over a specified phi range, the `sweep_data` routine must be used to acquire the phi data matrix. The user must call the module `set_collapse_info` prior to calling the `sweep_data` module in order to specify the resolution of the phi bins and to specify if the interleave option is to be utilized when building the phi matrix.

The parameter `exclude_dqual` holds a single value that is compared against the d_qual value found in the header record for the sensor being processed. If the user wishes to exclude data that is flagged with a specific d_qual value, the user should set the `exclude_dqual` value to this specific value. If the user wishes to include all data encountered, the user should set the `exclude_dqual` value to 255.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `read_drec` 1R
- `convert_to_units` 1R
- `sweep_discontinuous_data` 2R
- `sweep_mode_data` 2R
- `set_bin_info` 2R
- `center_and_band_values` 2R
- `fill_sensor_info` 2R
- `buffer_bin_fill` 2R
- `set_collapse_info` 2R
- `get_data_key` 1R
- `get_version_number` 1R
- `create_data_structure` 1R
- `create_idf_data_structure` 1R
- `ret_codes` 1H
- `libtrec_idfs` 2H

**BUGS**

None
EXAMPLES

Obtain data one sweep at a time from the virtual instrument RTLA, which is part of the
RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with
the TSS project.

```
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS ULONG data_key;
SDDAS USHORT vnum;
SDDAS FLOAT *ret_data, *ret_frac;
SDDAS LONG start_time_sec, start_time_nano, end_time_sec, end_time_nano;
SDDAS SHORT start_time_yr, start_time_day, end_time_yr, end_time_day;
SDDAS SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS CHAR *ret_bin, hdr_change, complete_acq;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}

get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = sweep_data (data_key, ",", vnum, idf_data_ptr, 1, &sen_numbers, &ret_data,
    &ret_frac, &ret_bin, &num_sen, &num_units, &data_block,
    &start_time_yr, &start_time_day, &start_time_sec,
    &start_time_nano, &end_time_yr, &end_time_day,
    &end_time_sec, &end_time_nano, &hdr_change, 255,
    &complete_acq);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by sweep_data routine.\n", status);
    exit (-1);
}
```
SWEEP_DISCONTINUOUS_DATA
function - returns sample-averaged sensor-specific data for data sets that roll over to a
minimum value when the maximum threshold has been reached

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT sweep_discontinuous_data (SDDAS_ULONG data_key,
   SDDAS_CHAR *exten, SDDAS_USHORT version,
   void *idf_data_ptr, SDDAS_LONG num_swps,
   SDDAS_SHORT **ret_sensors, SDDAS_FLOAT **ret_data,
   SDDAS_FLOAT **ret_frac, SDDAS_CHAR **bin_stat,
   SDDAS_SHORT *num_sen, SDDAS_SHORT **num_units,
   SDDAS_SHORT *block_size, SDDAS_SHORT *stime_yr,
   SDDAS_SHORT *stime_day, SDDAS_LONG *stime_sec,
   SDDAS_LONG *stimeNano, SDDAS_SHORT *etime_yr,
   SDDAS_SHORT *etime_day, SDDAS_LONG *etime_sec,
   SDDAS_LONG *etimeNano, SDDAS_CHAR *hdr_change,
   SDDAS_UCHAR exclude_dqual,
   SDDAS_CHAR *complete_acq)

ARGUMENTS
data_key   - unique value which indicates the data set of interest
exten    - two character extension to be added to IDFS file
         names when default files are not to be used, otherwise
         a null string
version    - IDFS data set identification number which allows for
         multiple openings of the same data set
idf_data_ptr   - pointer to the idf_data structure that temporarily
         holds sensor data and pertinent ancillary data for the
         data set of interest for a single sweep of data
num_swps   - the number of samples (sweeps) to average together
ret_sensors   - an array which holds the sensor number(s) for which
         data is returned
         - The array is initialized to -1 in all elements; valid
         sensor numbers start with 0
ret_data   - pointer to the data being returned (data for all sensors
         processed)
ret_frac   - pointer to the normalization factors for the data being
         returned
bin_stat   - pointer to status flags which are associated with each
data bin returned
         0 - no data has been placed into the data
         bin being processed
1 - data has been placed into the data bin being processed

num_sen - the number of elements in the ret_sensors array
num_units - an array holding the number of data sets to bypass in order to get to the data for the sensor being processed
block_size - the number of data values returned in a data buffer
stime_yr - the year value for the first sweep processed
stime_day - the day of year value for the first sweep processed
stime_sec - the time of day in seconds for the first sweep processed
stime_nano - the time of day residual in nanoseconds for the first sweep processed
etime_yr - the year value for the last sweep processed
etime_day - the day of year value for the last sweep processed
etime_sec - the time of day in seconds for the last sweep processed
etime_nano - the time of day residual in nanoseconds for the last sweep processed
hdr_change - flag which indicates a header change occurred while processing the data
   0 - a header change was not encountered during the processing of the data
   1 - a header change was encountered during the processing of the data
exclude_dqual - data is to be excluded if the d_qual flag associated with the data is set to the value specified
complete_acq - flag which indicates if all samples (sweeps) were acquired
   0 - not all samples were acquired
   1 - all samples were acquired
sweep_discontinuous_data - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SWEEP_DISCONTINUOUS_DATA

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWEEP_DISC_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>SWEEP_DISC_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call set_bin_info for this combination)</td>
</tr>
<tr>
<td>SWEEP_DISC_CENTER_BAND_MISSING</td>
<td>the routine center_and_band_values has not been called prior to calling the sweep_discontinuous_data routine</td>
</tr>
<tr>
<td>SWEEP_DISC_NO_PHI</td>
<td>data sets with PHI, MASS and/or CHARGE dimensions are not supported</td>
</tr>
<tr>
<td>FILL_ARRAY_MALLOC</td>
<td>no memory for structure which hold information pertinent to the sample-averaged data</td>
</tr>
<tr>
<td>FILL_DISC_MALLOC</td>
<td>no memory for fill_discontinuous structure</td>
</tr>
<tr>
<td>FILL_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>FILL_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FILL_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffers</td>
</tr>
<tr>
<td>FILL_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>FILL_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>FILL_DATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>SWEEP_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SWEEP_DATA_MALLOC</td>
<td>no memory for data buffer</td>
</tr>
<tr>
<td>SWEEP_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SWEEP_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SWEEP_SWP_MALLOC</td>
<td>no memory for sweep values in specified units</td>
</tr>
<tr>
<td>SWEEP_SWP_REALLOC</td>
<td>no memory for expansion of sweep values in specified units</td>
</tr>
<tr>
<td>DISC_DATA_MALLOC</td>
<td>no memory for the internal data buffers that are pertinent only to discontinuous data sets</td>
</tr>
<tr>
<td>SWEEP_DISC_WITH_FILL</td>
<td>the modules sweep_discontinuous_data and fill_discontinuous_data cannot be used interchangeably for the same data key, extension, version combination</td>
</tr>
<tr>
<td>BAD_VFMT</td>
<td>bad format character for variable width bin spacing</td>
</tr>
<tr>
<td>DISC_TMP_MALLOC</td>
<td>error codes returned by read_drec ()</td>
</tr>
<tr>
<td></td>
<td>error codes returned by convert_to_units ()</td>
</tr>
<tr>
<td></td>
<td>error codes returned by fill_sensor_info ()</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

Sweep_discontinuous_data is the IDFS sample-averaging read routine for discontinuous sensor-specific data, averaging num_swps sample sets (sweeps). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. Sweep_discontinuous_data processes sensor-specific data only, that is, it processes sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. If the instrument status (mode) data is desired, the user should use the sweep_mode_data routine. Sweep_discontinuous_data assumes that the data set of interest rolls over to a minimum value when the maximum threshold has been reached or to a maximum value when the minimum threshold has been reached. This assumption is crucial since multiple samples may be averaged together. Before each sample is added to the buffer, a check is made to see if a "boundary" or threshold has been crossed. If so, the value is adjusted so that the addition of the values result in a correct averaged value. Currently, these threshold values are preset at -180 (minimum threshold) and 180 (maximum threshold). If the data set does not roll over, the user should use the sweep_data routine. If the data set of interest is a combination of roll over and non-roll over data, for example, longitude data being returned along with science data, the user may use the sweep_discontinuous_data module in conjunction with the sweep_data routine, using the sweep_data routine to return the non-roll over data values and using the sweep_discontinuous_data routine to return the roll over data values. In order to do this correctly, the user must make use of multiple version numbers so that the same data files can be opened more than once. That is, use one version number for the non-roll over data...
and another version number for the roll over data. All IDFS routines that utilize a version number must be called once for each unique version number.

The data is processed one sweep at a time. Once the requested number of sweeps have been processed, the routine will return the data. If the requested number of sweeps could not be processed due to data acquisition problems (LOS_STATUS, NEXT_FILE_STATUS, EOF_STATUS), the routine will return the data and the normalization factors will reflect the number of sweeps processed so far. If more data is put online, the next call to the sweep_discontinuous_data routine will continue to accumulate data and will continue until the remaining sweeps have been acquired.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter idf_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the create_idf_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_idf_data_structure routine N times to create N instances of the idf_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set exten to a null string for real-time scenarios.

There are N many sub-buffers which hold the data in each of the requested data levels or units for each sensor. The user must process the data contained within these buffers before the next call to the sweep_discontinuous_data routine is made since the module will clear out these buffers for re-use if the requested number of sweeps were processed on the
previous call. The data values must be normalized using the normalization factors returned along with the data. The user is advised to check the value or values in the bin_stat array. If all values are 0, no data was placed into the buffer. This can happen if the sensors rotate or alternate when data is returned or if the data is excluded based upon d_qual or data cutoff values.

The size and spacing of the data buffer is either defined by the user or by elements contained within the virtual instrument definition document. The user must call the set_bin_info module before calling the sweep_discontinuous_data routine in order to specify how the binning of the data is to occur. In addition, the user must call the center_and_band_values module before calling the sweep_discontinuous_data module. If the sweep_discontinuous_data routine determines that no binning scheme has been selected, an error code is returned to the user.

The user should be aware that the data buffer that comes back from the sweep_discontinuous_data routine are NOT modified as far as missing bins is concerned. If the user wishes to fill in the missing bins according to the method specified in the call to the set_bin_info routine, the user must call the module buffer_bin_fill. If the data are collapsed over specified dimensions, the buffer_bin_fill module need not be called. The user should be advised that the sweep_discontinuous_data routine can not process data sets with a PHI, MASS and/or CHARGE dimensionality.

The default mode for the sweep_discontinuous_data routine is to return sensor data in raw units (no tables applied) for each of the sensors processed, with data cutoff values set at -3.0e38 (VALID_MIN) and 3.0e38 (VALID_MAX). The user may select the type of data, the units to be returned and the data cutoff values to be applied by calling the fill_sensor_info module prior to calling the sweep_discontinuous_data module. The user should make one call to the fill_sensor_info module for each sensor that is to be retrieved for each data type/units/data cutoff combination selected.

The parameter exclude_dqual holds a single value that is compared against the d_qual value found in the header record for the sensor being processed. If the user wishes to exclude data that is flagged with a specific d_qual value, the user should set the exclude_dqual value to this specific value. If the user wishes to include all data encountered, the user should set the exclude_dqual value to 255.

ERRORS
All errors within this routine are returned through the status variable. The include file ret_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The ret_codes.h file is described in section 1H of the IDFS Programmers Manual.

SEE ALSO
file_open 1R
read_drec 1R
convert_to_units 1R
Sweep discontinuous data one sweep at a time from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.

```c
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG start_time_sec, start_time_nano, end_time_sec, end_time_nano;
SDDAS_SHORT start_time_yr, start_time_day, end_time_yr, end_time_day;
SDDAS_SHORT status, *sen_numbers, num_sen, *num_units, data_block;
SDDAS_CHAR *ret_bin, hdr_change, complete_acq;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
generate_version_number (&vnum);
status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by create_idf_data_structure routine.\n", status);
}
```

**BUGS**

None

**EXAMPLES**

Obtain discontinuous data one sweep at a time from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.
exit (-1);

}  

status = sweep_discontinuous_data (data_key, "", vnum, idf_data_ptr, 1,  
    &sen_numbers, &ret_data, &ret_frac, &ret_bin, &num_sen,  
    &num_units, &data_block, &start_time_yr, &start_time_day,  
    &start_time_sec, &start_time_nano, &end_time_yr, &end_time_day,  
    &end_time_sec, &end_time_nano, &hdr_change, 255,  
    &complete_acq);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by sweep_discontinuous_data routine.\n", status);
    exit (-1);
}

Sweep_mode_data function - returns sample-averaged instrument status (mode) data

SYNOPSIS
#include "libtrec_idfs.h"
#include "ret_codes.h"

SDDAS_SHORT sweep_mode_data (SDDAS_ULONG data_key,
  SDDAS_CHAR *exten, SDDAS_USHORT version,
  void *idf_data_ptr, SDDAS_LONG num_swps,
  SDDAS_SHORT **ret_modes, SDDAS_FLOAT **ret_data,
  SDDAS_FLOAT **ret_frac, SDDAS_CHAR **bin_stat,
  SDDAS_SHORT *num_modes,
  SDDAS_SHORT **num_units, SDDAS_SHORT *block_size,
  SDDAS_SHORT *stime_yr, SDDAS_SHORT *stime_day,
  SDDAS_LONG *stime_sec, SDDAS_LONG *stime_nano,
  SDDAS_SHORT *etime_yr, SDDAS_SHORT *etime_day,
  SDDAS_LONG *etime_sec, SDDAS_LONG *etime_nano,
  SDDAS_CHAR *hdr_change, SDDAS_CHAR *complete_acq)

ARGUMENTS
  data_key         - unique value which indicates the data set of interest
  exten           - two character extension to be added to IDFS file names when
                   default files are not to be used, otherwise a null string
  version          - IDFS data set identification number which allows for multiple
                     openings of the same data set
  idf_data_ptr     - pointer to the idf_data structure that temporarily holds sensor
                     data and pertinent ancillary data for the data set of interest for
                     a single sweep of data
  num_swps         - the number of samples (sweeps) to average together
  ret_modes        - an array which holds the instrument status (mode) bytes for
                     which data is returned
                      - the array is initialized to -1 in all elements;
                      valid mode numbers start with 0
  ret_data         - pointer to the data being returned (data for all modes
                     processed)
  ret_frac         - pointer to the normalization factors for the data being returned
  bin_stat         - pointer to status flags which are associated with each data bin
                     returned
    0 - no data has been placed into the data bin being processed
    1 - data has been placed into the data bin being processed
  num_modes        - the number of elements in the ret_modes array
num_units - an array holding the number of data sets to bypass in order to get to the data for the instrument status (mode) value being processed

block_size - the number of data values returned in a data buffer

stime_yr - the year value for the first sweep processed

stime_day - the day of year value for the first sweep processed

stime_sec - the time of day in seconds for the first sweep processed

stime_nano - the time of day residual in nanoseconds for the first sweep processed

etime_yr - the year value for the last sweep processed

etime_day - the day of year value for the last sweep processed

etime_sec - the time of day in seconds for the last sweep processed

etime_nano - the time of day residual in nanoseconds for the last sweep processed

hdr_change - flag which indicates a header change occurred while processing the data

0 - a header change was not encountered during the processing of the data

1 - a header change was encountered during the processing of the data

complete_acq - flag which indicates if all samples (sweeps) were acquired

0 - not all samples were acquired

1 - all samples were acquired

sweep_mode_data - routine status (see TABLE 1)

**TABLE 1.** Status Codes Returned for **SWEEP_MODE_DATA**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWEEP_MODE_NOT_FOUND</td>
<td>the requested data_key, exten, version combination has no memory allocated for processing (user did not call file_open for this combination)</td>
</tr>
<tr>
<td>SWEEP_MODE_FILE_OPEN</td>
<td>the user did not request mode data processing when file_open was called</td>
</tr>
<tr>
<td>SWEEP_MODE_INFO_DUP</td>
<td>the requested data_key, exten, version combination has no memory allocated for the instrument status information</td>
</tr>
<tr>
<td>SWEEP_MODES_NOT_REQUESTED</td>
<td>the user did not call fill_mode_info for this combination</td>
</tr>
<tr>
<td>FILL_MODE_ARRAY_MALLOC</td>
<td>no memory for structure which hold information pertinent to the sample-averaged data</td>
</tr>
<tr>
<td>ALLOC_SMODE_INFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SMODE_UNITS_MALLOC</td>
<td>no memory to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SMODE_UNITS_REALLOC</td>
<td>no memory for expansion of space to hold the various data levels for the data buffer</td>
</tr>
<tr>
<td>SMODE_DATA_MALLOC</td>
<td>no memory for data buffer</td>
</tr>
<tr>
<td>SWEEP_MODE_WITH_FILL</td>
<td>the modules sweep_mode_data and fill_mode_data cannot be used interchangeably for the same data key, extension, version combination</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>
DESCRIPTION

Sweep_mode_data is the IDFS sample-averaging data read routine for instrument status (mode) values, averaging num_swps sample sets (sweeps). The data set of interest is referenced through the key value data_key which can be created using the get_data_key module. Sweep_mode_data processes instrument status data only. If sensor-specific data is desired, that is, sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle spacecraft potential and / or background data, the user should use the sweep_data / sweep_discontinuous_data routine(s).

The data is processed one sweep at a time. Once the requested number of sweeps have been processed, the routine will return the mode data. If the requested number of sweeps could not be processed due to data acquisition problems (LOS_STATUS, NEXT_FILE_STATUS, EOF_STATUS), the routine will return the data and the normalization factors will reflect the number of sweeps processed so far. If more data is put online, the next call to the sweep_mode_data routine will continue to accumulate data and will continue until the remaining sweeps have been acquired.

The parameter version allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the get_version_number routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.

The parameter idf_data_ptr is a pointer to the structure that is to hold all data pertinent to the data set being processed. The structure is created and the address to this structure is returned when a call to the create_idf_data_structure routine is made. The user also has the option of calling the module create_data_structure, which determines what type of data structure is needed for the IDFS data set of interest. In most cases, one data structure is sufficient to process any number of distinct data sets. However, if more than one structure is needed, the user may call the create_idf_data_structure routine N times to create N instances of the idf_data structure. The user must keep track of which pointer to send to the IDFS routines that utilize this structure.

If the file_open routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable exten. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable USER_DATA, which is set by the user, or in the user's home directory if the environment variable USER_DATA is not set. To open the default IDFS data files, exten should be set to a null
string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

There are N many sub-buffers which hold the data in each of the requested data levels or units for each mode. The user must process the data contained within these buffers before the next call to the `sweep_mode_data` routine is made since the module will clear out these buffers for re-use if the requested number of sweeps were processed on the previous call. The data values must be normalized using the normalization factors returned along with the data. The user is advised to check the value or values in the `bin_stat` array. If all values are 0, no data was placed into the buffer. This can happen if the status bytes rotate or alternate when data is returned.

In order to utilize the `sweep_mode_data` routine, the user must select the units to be returned and the data cutoff values to be applied by calling the `fill_mode_info` module prior to calling the `sweep_mode_data` module. The user should make one call to the `fill_mode_info` module for each instrument status byte that is to be retrieved for each units/data cutoff combination selected. If the `sweep_mode_data` routine determines that the `fill_mode_info` module was never called, an error code is returned.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- file_open 1R
- read_drec 1R
- convert_to_units 1R
- sweep_data 2R
- sweep_discontinuous_data 2R
- fill_mode_info 2R
- get_data_key 1R
- get_version_number 1R
- create_data_structure 1R
- create_idf_data_structure 1R
- ret_codes 1H
- libtrec_idfs 2H

**BUGS**

None

**EXAMPLES**

Obtain instrument status values one sweep at a time from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project.
#include "libtree_idfs.h"
#include "ret_codes.h"

SDDAS ULONG data_key;
SDDAS USHORT vnum;
SDDAS FLOAT *ret_data, *ret_frac;
SDDAS LONG start_time_sec, start_time_nano, end_time_sec, end_time_nano;
SDDAS SHORT start_time_yr, start_time_day, end_time_yr, end_time_day;
SDDAS SHORT status, *mode_numbers, num_modes, *num_units, data_block;
SDDAS CHAR *ret_bin, hdr_change, complete_acq;
void *idf_data_ptr;

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = create_idf_data_structure (&idf_data_ptr);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_idf_data_structure routine.\n", status);
    exit (-1);
}

status = sweep_mode_data (data_key, "", vnum, idf_data_ptr, 1, &mode_numbers,
    &ret_data, &ret_frac, &ret_bin, &num_modes, &num_units,
    &data_block, &start_time_yr, &start_time_day,
    &start_time_sec, &start_time_nano, &end_time_yr, &end_time_day,
    &end_time_sec, &end_time_nano, &hdr_change, &complete_acq);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by sweep_mode_data routine.\n", status);
    exit (-1);
}
UNITS_INDEX

function - returns index values to access the data returned by the time-averaging, sample-averaging, and spin-averaging modules for the data type/cutoff/units combination specified

SYNOPSIS

#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_SHORT units_index (SDDAS_ULONG data_key, SDDAS_CHAR *exten,
SDDAS_USHORT version, SDDAS_SHORT sensor,
SDDAS_FLOAT min, SDDAS_FLOAT max,
SDDAS_CHAR *tibs_to_apply, SDDAS_LONG *tbl_oper,
SDDAS_CHAR data_type, SDDAS_CHAR cal_set,
SDDAS_SHORT *units_ind, SDDAS_SHORT *num_units,
SDDAS_CHAR num_tbls)

ARGUMENTS

data_key - unique value which indicates the data set of interest
exten - two character extension to be added to IDFS file names when
default files are not to be used, otherwise a null string
version - IDFS data set identification number which allows for multiple
openings of the same data set
sensor - sensor identification number
min - the lower cutoff value for data that are to be put into the data buffers,
specified in terms of the units desired.
max - the upper cutoff value for data that are to be put into the data buffers,
specified in terms of the units desired.
tbls_to_apply - the tables that are to be applied in order to derive the desired units
tbl_oper - the operations that are to be applied to the specified tables in order to
derive the desired units
data_type - the type of data being requested
  1 - sensor data (SENSOR)
  2 - sweep step data (SWEEP_STEP)
  3 - calibration data (CAL_DATA)
  5 - data quality data (D_QUAL)
  6 - pitch angle data (PITCH_ANGLE)
  7 - start azimuthal angle data
     (START_AZ_ANGLE)
  8 - stop azimuthal angle data
     (STOP_AZ_ANGLE)
  9 - spacecraft potential data (SC_POTENTIAL)
 10 - background data (BACKGROUND)
cal_set - the calibration set from which requested calibration data
(CAL_DATA) is to be retrieved
- If calibration data is not being requested, this parameter is not utilized and it is suggested that the user pass a value of zero for this parameter.

**units_ind** - index value returned to access the correct sub-buffer returned from the time-averaging, sample-averaging, or spin-averaging routine for the data type/cutoff/units combination requested.

**num_units** - the number of units or data levels defined for the sensor in question (used as an index to get to the first buffer returned by the time-averaging or sample-averaging routine for the sensor in question).

**num_tbls** - the number of elements specified in the `tbls_to_apply` and `tbl_oper` parameters.

**units_index** - routine status (see Table 1).

### Table 1. Status Codes Returned for **UNITS_INDEX**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNITS_INDEX_NOT_FOUND</td>
<td>the requested data key, exten, version combination has no memory allocated for processing (user did not call <code>file_open</code> for this combination)</td>
</tr>
<tr>
<td>UNITS_NO_SENSOR</td>
<td>the requested sensor was not found amongst the defined data type/cutoff/units combinations (user did not call <code>fill_sensor_info</code> for this combination)</td>
</tr>
<tr>
<td>UNITS_NO_MATCH</td>
<td>the data type/cutoff/units combination requested was not found for the specified sensor</td>
</tr>
<tr>
<td>UNITS_INDEX_MODE_TYPE</td>
<td>instrument status (mode) data is not supported by the <code>units_index</code> routine</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

### Description

**Units_index** is the IDFS routine that returns index values that are used to access the data buffers returned by the IDFS routines that return time-averaged data (`fill_data` / `fill_discontinuous_data`), sample-averaged data (`sweep_data` / `sweep_discontinuous_data`), or spin-averaged data (`spin_data` / `spin_data_pixel`) for the sensor, data type / cutoff / units combination specified. The data set of interest is referenced through the key value `data_key` which can be created using the `get_data_key` module. The **units_index** module should be used for sensor-specific data only, that is, for sensor, sweep step, calibration, data quality, pitch angle, azimuthal angle, spacecraft potential and background data. If instrument status (mode) data is also being processed, the user should use the `mode_units_index` routine to retrieve index values to access the data buffers returned by the `fill_mode_data` / `sweep_mode_data` routine.

The parameter **version** allows multiple file openings for an IDFS data set. If the data, header and VIDF file for the specified data set need to be opened just once for processing, the same version number should be passed to all IDFS routines. However, for multiple file openings, the version number should be unique and all file manipulations performed by the IDFS routines will use the file descriptors defined for the version number specified. The user should call the `get_version_number` routine to retrieve a unique version number instead of choosing this value themselves. The retrieval of multiple data parameters from a single data source does not constitute the need for multiple version numbers; a single version number will suffice.
If the `file_open` routine is not to open the default set of IDFS files but a modified set of IDFS files, the two character extension applied to these data files must be supplied to this routine within the string variable `exten`. These files must have the identical name as the IDFS files with the two character identification code appended to the end of the file names (i.e. RTLA19922181432Dxx, RTLA19922181432Hxx, RTLA19922181432Ixx). The files must reside either in the directory specified by the environment variable `USER_DATA`, which is set by the user, or in the user's home directory if the environment variable `USER_DATA` is not set. To open the default IDFS data files, `exten` should be set to a null string. The usage of modified data sets is limited to post acquisition data; therefore, it is suggested that the user set `exten` to a null string for real-time scenarios.

The user may elect to call the `units_index` routine every time a return from the time-averaging, sample-averaging, or spin-averaging routine is made or may call the `units_index` routine once for each sensor, data type/cutoff/units combination requested and save the index values into variables for later usage. In either case, the call(s) to the `units_index` routine must be made after ALL calls to the `fill_sensor_info` routine have been made. The user may not have called the `fill_sensor_info` module if the default mode for the time-averaging, sample-averaging, or spin-averaging routine is sufficient. In this case, the user may retrieve the index values from the `units_index` routine only AFTER the first call to the time-averaging, sample-averaging, or spin-averaging routine has been made.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `ret_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `ret_codes.h` file is described in section 1H of the IDFS Programmers Manual.

**SEE ALSO**

- `file_open` 1R
- `fill_data` 2R
- `fill_discontinuous_data` 2R
- `fill_mode_data` 2R
- `sweep_data` 2R
- `sweep_discontinuous_data` 2R
- `sweep_mode_data` 2R
- `spin_data` 2R
- `spin_data_pixel` 2R
- `mode_units_index` 2R
- `fill_sensor_info` 2R
- `get_data_key` 1R
- `get_version_number` 1R
- `ret_codes` 1H
- `user_def` 1H
- `libtrec_idfs` 2H
BUGS
None

EXAMPLES
Retrieve the index values to access data that is returned for sensor 1 from the virtual instrument RTLA, which is part of the RETE instrument/experiment, which is part of the TSS-1 mission, which is identified with the TSS project. Assume that there is one table applicable to this virtual instrument.

```
#include "libtrec_idfs.h"
#include "ret_codes.h"
#include "user_defs.h"

SDDAS_ULONG data_key;
SDDAS_USHORT vnum;
SDDAS_FLOAT sen_min, sen_max;
SDDAS_LONG tbl_oper[1];
SDDAS_SHORT uind_base, status, sen_units, sensor;
SDDAS_CHAR tbls_to_apply[1], num_tbls;

sen_min = VALID_MIN;
sen_max = VALID_MAX;
sensor = 1;
num_tbls = 1;
tbls_to_apply[0] = 0;
tbl_oper[0] = 0;        /* look-up operation */

status = get_data_key ("TSS", "TSS-1", "RETE", "RETE", "RTLA", &data_key);
if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by get_data_key routine.\n", status);
    exit (-1);
}
get_version_number (&vnum);

status = units_index (data_key, ",", vnum, sensor, sen_min, sen_max,
                    tbls_to_apply, tbl_oper, SENSOR, 0, &uind_base,
                    &sen_units, num_tbls);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by units_index routine.\n", status);
    exit (-1);
}
```
CREATE_SCF_DATA_STRUCTURE
function - creates an instance of the scf_data structure

SYNOPSIS
#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT create_scf_data_structure (SDDAS_CHAR *filename,
                                     SDDAS_USHORT scf_version,
                                     void **scf_data_ptr)

ARGUMENTS
filename           - the name of the SCF file of interest
scf_version        - SCF identification number which allows for multiple
                    openings of the same SCF file
scf_data_ptr       - pointer to the newly created scf_data structure
create_scf_data_structure - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for CREATE_SCF_DATA_STRUCTURE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for</td>
</tr>
<tr>
<td></td>
<td>processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_CREATE_ALL_MALLOC</td>
<td>no memory to hold the address of all allocated scf_data structures</td>
</tr>
<tr>
<td>SCF_CREATE_ALL_REALLOC</td>
<td>no memory for expansion of the area that holds the address of all allocated</td>
</tr>
<tr>
<td></td>
<td>scf_data structures</td>
</tr>
<tr>
<td>SCF_CREATE_MALLOC</td>
<td>no memory for the scf_data structure</td>
</tr>
<tr>
<td>SCF_OUTPUT_MALLOC</td>
<td>no memory for the information pertaining to the values for the output variables</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION
Create_scf_data_structure creates an instance of the scf_data structure that is used by the SCF software to return the results from the execution of the SCF algorithm. With each call to this module, a new scf_data structure is created and the address of this structure is returned. In order to access the elements within the scf_data structure, the user must explicitly cast the returned void pointer to a pointer of the type struct scf_data. Before the call to the create_scf_data_structure module can be made, a call to the routine scf_open with the identical filename and scf_version parameters must be made; otherwise, an error code is returned. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing
this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

Since the SCF file dictates the number of output variables and the dimensionality of these variables, the user should call the `create_scf_data_structure` routine once for each distinct SCF file being processed. When multiple instances of the `scf_data` pointer are created, it is the responsibility of the user to keep track of which pointer to send to the SCF routines that utilize this structure. The contents of this structure is described in section 3S of the IDFS Programmers Manual.

The address and associated memory of each `scf_data` structure that is created can be freed through the `free_scf_info` routine. The user must not free the memory themselves since the SCF software will attempt to free the memory location and the result is uncertain.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- `scf_open`  3R
- `free_scf_info`  3R
- `SCF_codes`  3H
- `libbase_SCF`  3H
- `scf_data`  3S

**BUGS**

None

**EXAMPLES**

Create one instance of the `scf_data` structure that is to be associated with the SCF file `TMMO_EXAMPLE` and return the address in the specified parameter. Cast the returned void pointer so that elements of the `scf_data` structure can be referenced.

```c
#include "libbase_SCF.h"
#include "SCF_codes.h"

struct scf_data *SCF_DATA;
SDDAS_USHORT scf_vnum;
SDDAS_SHORT status;
void *scf_data_ptr;

scf_version_number (&scf_vnum);
```
status = create_scf_data_structure ("TMMO_EXAMPLE", scf_vnum, &scf_data_ptr);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by create_scf_data_structure routine.\n", status);
    exit (-1);
}

SCF_DATA = (struct scf_data *) scf_data_ptr;

/* Print the name of the SCF file associated with this scf_data structure. */

printf ("\n SCF filename = %s", SCF_DATA->filename);
FREE_SCF_INFO
    function - frees all the memory allocated by the SCF routines

SYNOPSIS
    #include "libbase_SCF.h"

    void free_scf_info (void)

ARGUMENTS
    No arguments for this routine

DESCRIPTION
    Free_scf_info frees all memory that has been allocated by the SCF routines. The computer operating system normally takes care of freeing any memory before terminating the program; however, for a clean exit, the user should call this module before exiting from the program. In addition, the user may call this module if a total restart of the SCF software is desired without restarting the program. In the case of a total restart, the user is advised to call the module init_scf before any other SCF routine since the free_scf_info routine merely frees allocated memory; it does not re-initialize variables used by the SCF software.

    If any scf_data structures were created using the create_scf_data_structure routine, the free_scf_info module will free the memory associated with elements contained in the scf_data structure and the data structure itself. The user must not attempt to free this memory since the SCF software will also attempt to free the memory.

ERRORS
    This routine returns no status or error codes.

SEE ALSO
    init_scf  3R
    create_scf_data_structure  3R
    libbase_SCF  3H
    scf_data  3S

BUGS
    None

EXAMPLES
    The usage of this routine is quite simple since no parameters are needed:

    #include "libbase_SCF.h"

    free_scf_info ();
INIT_SCF
    function - initializes the system for SCF processing

SYNOPSIS
    #include "libbase_SCF.h"

    void init_scf (void)

ARGUMENTS
    No arguments for this routine

DESCRIPTION
    Init_scf initializes the system prior to the processing of the information contained in the SCF files. A call must be made to this routine before any other SCF routines are invoked.

Since the SCF data access software must interface with the database, calls must be made to the dbInitialize and CfgInit modules when the init_scf module is called. The user is referred to the webpages http://cluster/libdbSQL.html and http://cluster/libCfg.html for an explanation of these routines.

ERRORS
    This routine returns no status or error codes.

BUGS
    None

EXAMPLES
    The usage of this routine is quite simple since no parameters are needed:

    #include "libbase_SCF.h"

    CfgInit ();
    dbInitialize ();
    init_scf ();
LOAD_SCF

function - loads the contents of the SCF file

SYNOPSIS

#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT load_scf (SDDAS_CHAR *filename, SDDAS_USHORT scf_version,
                     SDDAS_SHORT btime_yr, SDDAS_SHORT btime_day,
                     SDDAS_LONG btime_sec, SDDAS_LONG btime_nano,
                     SDDAS_SHORT etime_yr, SDDAS_SHORT etime_day,
                     SDDAS_LONG etime_sec, SDDAS_LONG etime_nano)

ARGUMENTS

filename - the name of the SCF file of interest
scf_version - SCF identification number which allows for multiple
              openings of the same SCF file
btime_yr - the start year for IDFS data access
btime_day - the start day of year for IDFS data access
btime_sec - the start time of day in seconds for IDFS data access
btime_nano - the start time of day residual in nanoseconds
etime_yr - the end year for IDFS data access
etime_day - the end day of year for IDFS data access
etime_sec - the end time of day in seconds for IDFS data access
etime_nano - the end time of day residual in nanoseconds
load_scf - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for LOAD_SCF

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_SCF_FILE</td>
<td>error opening the specified SCF file</td>
</tr>
<tr>
<td>LOCATE_SCF_MALLOC</td>
<td>no memory for SCF definition structure</td>
</tr>
<tr>
<td>LOCATE_SCF_REALLOC</td>
<td>no memory for expansion of SCF definition structure</td>
</tr>
<tr>
<td>SCF_CONTACT_MALLOC</td>
<td>no memory for contact information contained in SCF file</td>
</tr>
<tr>
<td>SCF_COMMENTS_MALLOC</td>
<td>no memory for comment information contained in SCF file</td>
</tr>
<tr>
<td>SCF_INPUTS_MALLOC</td>
<td>no memory for SCF input variable structure</td>
</tr>
<tr>
<td>SCF_INPUT_TBL_MALLOC</td>
<td>no memory for unit conversion information for input variables</td>
</tr>
<tr>
<td>SCF_TEMP_MALLOC</td>
<td>no memory for SCF temporary variable structure</td>
</tr>
<tr>
<td>SCF_OUTPUT_VAR_MALLOC</td>
<td>no memory for SCF output variable structure</td>
</tr>
<tr>
<td>SCF_EONS_MALLOC</td>
<td>no memory for equations structure</td>
</tr>
<tr>
<td>SCF_EONS_REALLOC</td>
<td>no memory for expansion of equations structure</td>
</tr>
<tr>
<td>SCF_ARGS_MALLOC</td>
<td>no memory for the arguments/operands specified in the equations</td>
</tr>
<tr>
<td>SCF_INDEX_MALLOC</td>
<td>no memory for index variable information</td>
</tr>
<tr>
<td>SCF_MAP_MALLOC</td>
<td>no memory for the map of all defined variables</td>
</tr>
<tr>
<td>SCF_VOID</td>
<td>the function specified is a void function – no resultant variable should be specified</td>
</tr>
<tr>
<td>SCF_NON_VOID</td>
<td>the function specified is a non-void function – the resultant variable is missing from the equation</td>
</tr>
<tr>
<td>SCF_NUM_ARGS</td>
<td>incorrect number of arguments specified for function call</td>
</tr>
</tbody>
</table>
STATUS CODE | EXPLANATION OF STATUS
--- | ---
SCF_RES_LENGTH | invalid length for the resultant variable
SCF_ARG_RANK | the dimension of the argument/operand is invalid for the function/operator specified
SCF_RES_RANK | the dimension of the resultant variable does not match the dimension returned by the function/operator specified
SCF_NO_INDEX | the argument specified must not be an indexed variable
VIDF_OPEN_PTR_MALLOC | no memory for IDFS location pointers
VIDF_OPEN_EX_REALLOC | no memory for experiment definition structure expansion
NO_DATA | there is no VIDF available for the requested time period
SCF_SIZE_MISMATCH | the matrix size for all dimensions must be the same for the resultant and the arguments in the equation
SCF_MASK_LENGTHS | invalid dimension sizes for the resultant and arguments in the equation
SCF_SQUARE_ARG | the argument matrix in the equation is not a square matrix
SCF_SQUARE_RES | the resultant matrix in the equation is not a square matrix
SCF_AORDER_MISMATCH | the order values are not compliant for the arguments in the equation
SCF_RORDER_MISMATCH | the order value for the resultant is not compliant with arguments in the equation
SCF_NO_FUNCTION | the function being requested is not a registered function
SCF_DIMEN_MALLOC | no memory for the multi-dimensional data array
SCF_TENSOR_MANY_ARGS | there are more than 10 arguments defined for the tensor function
SCF_TENSOR_SAME_RANK | the resultant and the arguments must be the same rank for the tensor function used
SCF_TSIZE_MISMATCH | the matrix size of the resultant is invalid for the selected matrix operation
ALL_OKAY | the routine terminated successfully

Load_scf utilizes the file_open and get_data_key IDFS read routines. For a complete listing of the error codes returned by these modules, the user is referred to section 1R of the IDFS Programmers Manual.

DESCRIPTION

Load_scf opens and loads the contents of the SCF file. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. This routine needs a start and stop time in order to retrieve information from the VIDF file for the IDFS data sets utilized as input variables. This routine is used when the user is only concerned with accessing the contents of the SCF, not with the execution of the algorithm contained in the SCF file. If the user intends to execute the algorithm, the user should use the scf_open routine which calls the load_scf routine in addition to opening the IDFS data sets that are pertinent to the SCF file being processed. In either case, once the contents of the SCF file has been loaded, the user may call the read_scf routine to retrieve information from the SCF file.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source
does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

ERRORS
All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.

SEE ALSO
scf_open 3R
read_scf 3R
scf_version_number 3R
SCF_codes 3H
libbase_SCF 3H

BUGS
None

EXAMPLES
Retrieve the contents of the SCF file TMMO_EXAMPLE.

#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_LONG btime_sec, btime_nsec, etime_sec, etime_nsec;
SDDAS_USHORT scf_vnum;
SDDAS_SHORT status, btime_yr, btime_day, etime_yr, etime_day;

btime_yr = 1992;
btime_day = 217;
btime_sec = 32340;
btime_nsec = 0;
etime_yr = 1992;
etime_day = 217;
etime_sec = 32342;
etime_nsec = 0;

csf_version_number (&scf_vnum);
status = load_scf("TMMO_EXAMPLE", scf_vnum, btime_yr, btime_day, btime_sec, btime_nsec, etime_yr, etime_day, etime_nsec);
if(status != ALL_OKAY)
{
    printf("\n Error %d returned by load_scf routine.\n", status);
    exit (-1);
}
READ_SCF
function - retrieve information from the SCF file

SYNOPSIS
#include "libbase_SCF.h"
#include "SCF_file_defs.h"
#include "SCF_codes.h"

SDDAS_SHORT read_scf (SDDAS_CHAR *filename, SDDAS_USHORT scf_version,
SDDAS_LONG field, SDDAS_LONG which_src,
SDDAS_CHAR *var)

ARGUMENTS
filename   - the name of the SCF file of interest
scf_version   - SCF identification number which allows for multiple
openings of the same SCF file
field    - specified field in the SCF file
which_src   - the variable or equation definition from which the
required field is to be retrieved
var    - output value(s) associated with the selected field
read_scf   - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for READ_SCF

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open or load_scf for this combination)</td>
</tr>
<tr>
<td>READ_SCF_BAD_INPUT</td>
<td>invalid input variable reference number (bad value for which_src parameter)</td>
</tr>
<tr>
<td>READ_SCF_BAD_DSRC</td>
<td>error encountered for the IDFS data source specified for the input variable</td>
</tr>
<tr>
<td>READ_SCF_BAD_FIELD</td>
<td>the information requested is not relevant for the variable referenced</td>
</tr>
<tr>
<td>READ_SCF_BAD_TEMP</td>
<td>invalid temporary variable reference number (bad value for which_src parameter)</td>
</tr>
<tr>
<td>READ_SCF_NO_DIMEN</td>
<td>dimension length(s) is not pertinent for scalar quantities</td>
</tr>
<tr>
<td>READ_SCF_BAD_OUTPUT</td>
<td>invalid output variable reference number (bad value for which_src parameter)</td>
</tr>
<tr>
<td>READ_SCF_BAD_EQNS</td>
<td>invalid equation number (bad value for which_src parameter)</td>
</tr>
<tr>
<td>READ_SCF_ELSE_INFO</td>
<td>the information requested is not pertinent to the equation (there is no ELSE component for the equation in question)</td>
</tr>
<tr>
<td>READ_SCF_BAD_FUNCTION</td>
<td>invalid function used in the equation number referenced</td>
</tr>
<tr>
<td>READ_SCF_BAD_INDEX</td>
<td>invalid variable name as index value in the equation referenced</td>
</tr>
<tr>
<td>READ_SCF_BAD_TOKEN</td>
<td>invalid variable name in the equation number reference</td>
</tr>
<tr>
<td>READ_SCF_NO_TOKEN</td>
<td>the field being requested is not defined</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION
Read_scf returns data for a selected field within the SCF file. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. The value of interest is indicated through the field number (field). A list of field numbers together with a set of built-in acronyms which can be used as input to this routine...
are found in the `SCF_file_defs.h` file. This file is described in section 3H of the IDFS Programmers Manual. If the field being requested is associated with an input variable, temporary variable, output variable or equation definition, the variable or equation from which the data is to be retrieved is indicated through the parameter `which_src`. If the field being requested is not associated with a variable or equation definition, this parameter value is ignored by this routine; therefore, any value can be passed in for this parameter (the acronym NOT_USED is suggested).

The parameter `scf_version` allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the `scf_version_number` routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The routine returns data through the variable `var`. The variable `var` should be of the format of the data being requested (e.g., SDDAS_FLOAT, SDDAS_LONG, SDDAS_CHAR, etc.) and is cast as a character pointer when input into the routine. If the field being returned is an array field, `var` must be of sufficient size to hold the entire length of the data requested. The routine does not know internally whether a requested variable is an array or a single variable. This determination and the appropriate action must be taken by the calling routine. Prior to calling the `read_scf` routine, a call to either the `scf_open` or `load_scf` routine must have been made with the same `filename` and `scf_version` designations to open and load the contents of the appropriate SCF file.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- `scf_open` 3R
- `load_scf` 3R
- `scf_version_number` 3R
- `SCF_file_defs` 3H
- `SCF_codes` 3H
- `libbase_SCF` 3H

**BUGS**

None
EXAMPLES
Obtain the number of equations defined in the SCF file TMMO_EXAMPLE. The number of equations is returned in the variable num_of. Since this field does not pertain to an individual variable or equation, the acronym NOT_USED is passed for the which_src parameter. Once that information is known, retrieve the equations, one at a time.

```
#include "libbase_SCF.h"
#include "SCF_file_defs.h"
#include "SCF_codes.h"

register SDDAS_LONG i;
SDDAS_UHShORT scf_vnum;
SDDAS_LONG btime_sec, btime_nsec, etime_sec, etime_nsec, num_of;
SDDAS_SHORT status, btime_yr, btime_day, etime_yr, etime_day;
SDDAS_CHAR string[90];

btime_yr = 1992;
btime_day = 217;
btime_sec = 32340;
btime_nsec = 0;
etime_yr = 1992;
etime_day = 217;
etime_sec = 32342;
etime_nsec = 0;

scf_version_number (&scf_vnum);
status = load_scf ("TMMO_EXAMPLE", scf_vnum, btime_yr, btime_day, btime_sec,
                  btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by load_scf routine.n", status);
    exit (-1);
}

status = read_scf ("TMMO_EXAMPLE", scf_vnum, S_NUM_EQNS, NOT_USED,
                   (SDDAS_CHAR *) &num_of);
if (status != ALL_OKAY)
{
    printf ("n Error %d returned by read_scf routine.n", status);
    exit (-1);
}

for (i = 0; i < num_of; ++i)
{
    status = read_scf ("TMMO_EXAMPLE", scf_vnum, S_EQUATION, i,
                       (SDDAS_CHAR*) string);
}
if (status != ALL_OKAY)
{
  printf ("\n Error %d returned by read_scf routine.\n", status);
  exit (-1);
}
printf ("Equation %ld : %s\n", i, string);
}
SCF_OPEN
function - loads the contents of the SCF file and opens all IDFS data sets identified as input variables

SYNOPSIS
#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_open (SDDAS_CHAR *filename, SDDAS_USHORT scf_version,
SDDAS_SHORT btime_yr, SDDAS_SHORT btime_day,
SDDAS_LONG btime_sec, SDDAS_LONG btime_nano,
SDDAS_SHORT etime_yr, SDDAS_SHORT etime_day,
SDDAS_LONG etime_sec, SDDAS_LONG etime_nano)

ARGUMENTS
filename  - the name of the SCF file of interest
scf_version  - SCF identification number which allows for multiple
openings of the same SCF file
btime_yr  - the year at which algorithm execution is to commence
btime_day  - the day of year at which algorithm execution is to commence
btime_sec  - the time of day in seconds at which algorithm execution is to commence
btime_nano  - the time of day residual in nanoseconds
etime_yr  - the year at which algorithm execution is to terminate
etime_day  - the day of year at which algorithm execution is to terminate
etime_sec  - the time of day in seconds at which algorithm execution is to terminate
etime_nano  - the time of day residual in nanoseconds
scf_open  - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_OPEN

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCF_OPEN_RTIME</td>
<td>the SCF software does not support real-time processing</td>
</tr>
<tr>
<td>SCF_MATRIX_MALLOC</td>
<td>no memory for the data matrix that holds data for all defined variables</td>
</tr>
<tr>
<td>SCF_OPEN_ERROR</td>
<td>error returned from call to file_open for specified input variables</td>
</tr>
<tr>
<td>SCF_ALLOC_PLOT_LOC</td>
<td>no memory for structure that holds timing information</td>
</tr>
<tr>
<td>SCF_REALLOC_PLOT_LOC</td>
<td>no memory for expansion of structure that holds timing information</td>
</tr>
<tr>
<td>SCF_FRAC_MALLOC</td>
<td>no memory for normalization factors for the data for the input variables</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by file_open ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by get_data_key ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by create_idf_data_structure ()</td>
</tr>
<tr>
<td></td>
<td>Error codes returned by load_scf ()</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the routine terminated successfully</td>
</tr>
</tbody>
</table>
Scf_open utilizes the file_open, get_data_key and create_idf_data_structure IDFS read routines. For a complete listing of the error codes returned by these modules, the user is referred to section 1R of the IDFS Programmers Manual.

DESCRIPTION

Scf_open opens and loads the contents of the SCF files and all referenced IDFS data sets. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. Once the contents of the specified SCF file is loaded, an attempt is made to open the IDFS data set(s) that are specified as the sources for the input variables. The SCF file itself has no dependence on time; that is, the algorithm can be applied to data taken at any time. However, the IDFS data files that are opened, the header, data, and VIDF file, are dependent on the time range specified. The appropriate IDFS data files are searched for within the current on-line database. If the files do exist on the local machine, the files are opened. If the files do not exist on the local machine, an error code is returned since this routine does not autopromote needed, but off-line, data. This routine opens the first set of IDFS data files within the time span over which data is to be processed. If there is more than one file set within the requested time interval, the remaining IDFS files will be opened and processed after the currently opened files are processed.

The SCF software does not support real-time processing. In the real-time scenario, the header and data files are incomplete and it is possible to attempt to read from either file prior to the data being received. Therefore, the values for the input variables may not be attainable when the algorithm is being executed and thus, the algorithm cannot be executed correctly. In the playback scenario, the data is always available provided data was collected at the time period being processed.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. In either case, the specified SCF will only be opened once for each unique parameter set. If additional calls are made to this routine with the same parameter set, the module simply returns the ALL_OKAY status code. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

ERRORS

All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.
SEE ALSO
    load_scf  3R
    scf_version_number  3R
    SCF_codes  3H
    libbase_SCF  3H

BUGS
    None

EXAMPLES
    Retrieve the contents of the SCF file TMMO_EXAMPLE and open the associated IDFS data sets for the time range specified.

    #include "libbase_SCF.h"
    #include "SCF_codes.h"

    SDDAS_LONG btime_sec, btime_nsec, etime_sec, etime_nsec;
    SDDAS_USHORT scf_vnum;
    SDDAS_SHORT status, btime_yr, btime_day, etime_yr, etime_day;

    btime_yr = 1992;
    btime_day = 217;
    btime_sec = 32340;
    btime_nsec = 0;

    etime_yr = 1992;
    etime_day = 217;
    etime_sec = 32342;
    etime_nsec = 0;

    scf_version_number (&scf_vnum);
    status = scf_open ("TMMO_EXAMPLE", scf_vnum, btime_yr, btime_day, btime_sec,
                        btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec);
    if (status != ALL_OKAY)
    {
        printf ("n Error %d returned by scf_open routine.\n", status);
        exit (-1);
    }
SCF_OUTPUT_DATA

function - execute the algorithm defined in the SCF file and return the values for the output variables in the specified scf_data structure

SYNOPSIS

#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_output_data (SDDAS_CHAR *filename,
                              SDDAS_USHORT scf_version, void *scf_data_ptr)

ARGUMENTS

filename   - the name of the SCF file of interest
scf_version   - SCF identification number which allows for multiple openings of the same SCF file
scf_data_ptr   - pointer to the scf_data structure that is to hold the values that are returned
scf_output_data  - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_OUTPUT_DATA

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_OUTPUT_CALC</td>
<td>the data accumulation rate has not been set (user did not call scf_sample_rate for this combination)</td>
</tr>
<tr>
<td>SCF_OUTPUT_DATA_STR</td>
<td>the scf_data structure is not associated with the specified SCF file</td>
</tr>
<tr>
<td>SCF_INVALID_INDEX</td>
<td>the array index value computed at execution time is invalid</td>
</tr>
<tr>
<td>SCF_BAD_LOGICAL_OPER</td>
<td>invalid logical operator specified in the IF-ELSE-ENDIF construct</td>
</tr>
<tr>
<td>SCF_FAST_BAD_LOCATE</td>
<td>an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>SCF_PROCESS_BAD_EX</td>
<td>an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>SCF_BAD_FRAC</td>
<td>invalid normalization factor computed</td>
</tr>
<tr>
<td>SCF_ACQ_MANY_READS</td>
<td>the acquisition time is incorrect for the vector IDFS source selected as the control variable</td>
</tr>
<tr>
<td>SCF_TERMINATE</td>
<td>processing must stop due to data not being on-line</td>
</tr>
<tr>
<td>SCF_NO_FUNCTION</td>
<td>the function being requested is not a registered function</td>
</tr>
<tr>
<td>SCF_NO_LIBRARY</td>
<td>the shared object library which holds the user-defined function can not be opened</td>
</tr>
<tr>
<td>SCF_NO_FUNC_IN_LIB</td>
<td>the user-defined function is not found in the specified shared object library</td>
</tr>
<tr>
<td>SCF_TDIMEN</td>
<td>invalid value for the dimension argument specified in the tensor summation equation</td>
</tr>
<tr>
<td>SCF_TSUM_VDIMEN</td>
<td>invalid dimension sizes for resultant row/column vector in the tensor summation equation</td>
</tr>
<tr>
<td>SCF_TSUM_ROW_DIMEN</td>
<td>cannot collapse over the first dimension since the argument is a row vector in the tensor summation equation</td>
</tr>
<tr>
<td>SCF_TSUM_COL_DIMEN</td>
<td>cannot collapse over the second dimension since the argument is a column vector in the tensor summation equation</td>
</tr>
<tr>
<td>SCF_TSUM_RSIZE</td>
<td>the size of the resultant tensor is incorrect for summing over the requested dimension in the tensor summation equation</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>SCF_TWDIMEN</td>
<td>invalid value for the dimension argument specified in the tensor weighted summation equation</td>
</tr>
<tr>
<td>SCF_TWSUM_VDIMEN</td>
<td>invalid dimension sizes for resultant row/column vector in the tensor weighted summation equation</td>
</tr>
<tr>
<td>SCF_TWSUM_ROW_DIMEN</td>
<td>cannot collapse over the first dimension since the argument is a row vector in the tensor weighted summation equation</td>
</tr>
<tr>
<td>SCF_TWSUM_COL_DIMEN</td>
<td>cannot collapse over the second dimension since the argument is a column vector in the tensor weighted summation equation</td>
</tr>
<tr>
<td>SCF_TWSUM_RSIZE</td>
<td>the size of the resultant tensor is incorrect for summing over the requested dimension in the tensor weighted summation equation</td>
</tr>
<tr>
<td>SCF_TW_WLEN</td>
<td>the size of the array of weight factors is incorrect for collapsing over the requested dimension in the tensor weighted summation equation</td>
</tr>
<tr>
<td>SCF_TSPACE</td>
<td>invalid value for the bin_spacing argument specified in the tensor integral equation</td>
</tr>
<tr>
<td>SCF_TINT_CLEN</td>
<td>the size of the array of center values in incorrect for collapsing over the requested dimension in the tensor integral equation</td>
</tr>
<tr>
<td>SCF_TINSERT_SDIMEN</td>
<td>invalid dimension sizes for the start / stop index arguments in the tensor insertion equation</td>
</tr>
<tr>
<td>SCF_TINSERT_START</td>
<td>the start index value is greater than the stop index value in the tensor insertion equation</td>
</tr>
<tr>
<td>SCF_TINSERT_INDEX</td>
<td>invalid start / stop index values specified in the tensor insertion equation</td>
</tr>
<tr>
<td>SCF_TINSERT_SIZE</td>
<td>the number of elements to be inserted does not match the size defined by the start / stop index values in the tensor insertion equation</td>
</tr>
<tr>
<td>SCF_TEXTRACT_SDIMEN</td>
<td>invalid dimension sizes for the start / stop index arguments in the tensor extraction equation</td>
</tr>
<tr>
<td>SCF_TEXTRACT_START</td>
<td>the start index value is greater than the stop index value in the tensor extraction equation</td>
</tr>
<tr>
<td>SCF_TEXTRACT_INDEX</td>
<td>invalid start / stop index values specified in the tensor extraction equation</td>
</tr>
<tr>
<td>SCF_TEXTRACT_RES_RANK</td>
<td>the dimension (rank) of the resultant is inconsistent with the start / stop index values specified in the tensor extraction equation</td>
</tr>
<tr>
<td>SCF_TEXTRACT_RES_DIMEN</td>
<td>invalid dimension sizes for the resultant in the tensor extraction equation</td>
</tr>
<tr>
<td>SCF_BREAK_STMT</td>
<td>a BREAK statement was specified outside of a FOR loop</td>
</tr>
<tr>
<td>SCF_FILL_SZ</td>
<td>the size of the array that holds the data once it has been converted into the requested units is not large enough to hold the data that is being processed</td>
</tr>
<tr>
<td>SCF_TENSOR_VECTOR_SRC</td>
<td>Multi-dimensional IDFS data source cannot serve as a controller</td>
</tr>
<tr>
<td>CUR_TIME_NOT_FOUND</td>
<td>an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>CHK_TDATA_NOT_FOUND</td>
<td>an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>CHK_DATA_NOT_FOUND</td>
<td>an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed</td>
</tr>
</tbody>
</table>

Error codes returned by `read_drec ()`
Error codes returned by `convert_to_units ()`
Error codes returned by `reset_experiment_info ()`
Error codes returned by `file_pos ()`
Error codes returned by `next_file_start_time ()`
Error codes returned by `create_idf_data_structure ()`

**Scf_output_data** utilizes the `convert_to_units`, `read_drec`, `reset_experiment_info` and `file_pos` IDFS read routines. For a complete listing of the error codes returned by these modules, the user is referred to section 1R of the IDFS Programmers Manual.
DESCRIPTION

**Scf_output_data** returns data for all the defined output variables evaluated during the current time step of the algorithm. The SCF file of interest is referenced through the parameter **filename**. The **filename** parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. Due to the nature of the processing, it is possible that the current time interval may cross a file boundary; that is, the start time of the interval is within one file and the end time of the interval is within the next file. If the next file is not available, the status code **SCF_TERMINATE** will be returned. Subsequent calls to the **scf_output_data** routine will continue to return **SCF_TERMINATE** until the calling module terminates. Therefore, if the user does not look for the **SCF_TERMINATE** status code upon return from this module and terminate processing appropriately, the program will end up in an infinite loop. This status code may be returned when the user-requested end time is located between the start and stop time period for the current iteration of the algorithm. In this case, the data for the controlling data set will be complete, but the acquisition of other sets of data required by the SCF may be incomplete due to data files not being available (online) to complete the acquisition. It is up to the user whether or not to utilize the sample. This status code may also be returned in the midst of processing if the data file became unavailable (off-line) after processing of the data began. If this happens, it is not known whether the controlling data set or the other data sets ran into the problem; therefore, it is best to simply throw away the data and terminate processing. Therefore, appropriate termination must be deciphered by the user program.

The returned data is placed in the **scf_data** structure that is referenced by the argument **scf_data_ptr**. The argument **scf_data_ptr** is a pointer to the structure that is to hold all data pertinent to the SCF file being processed. The structure is created and the address to this structure is returned when a call to the **create_scf_data_structure** routine is made. The contents of this structure is described in section 3S of the IDFS Programmers Manual. Since the SCF file dictates the number of output variables and the dimensionality of these variables, the user should call the **create_scf_data_structure** routine once for each distinct SCF file being processed and this pointer should be passed in conjunction with the named SCF file when the output variable values are being retrieved. If the **scf_output_data** routine determines that the **scf_data** structure being referenced is not associated with the named SCF file, an error code is returned.

The parameter **scf_version** allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the **scf_version_number** routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.
The number of output variables returned is always indicated in the `scf_data` structure in the `num_output` element. The data that is returned is referenced by the `output_data` element of the `scf_data` structure. This element is a pointer to the memory that holds all values for all output variables. In order to get to the data for a specific output variable, the user must retrieve an index into this memory block. These indexes are returned in the `output_index` element of the `scf_data` structure, which is an array of values, with one value being returned per output variable. Indexing into this array starts at position zero. Once this index value has been retrieved, the user can then reference the data for a particular output variable.

The output variables returned may be of different dimensionalities; that is, the data being returned may be a combination of scalar, vector or tensor quantities since the SCF software supports multi-dimensional output quantities (up to 10-D), e.g. `OUTPUT1[n][n][n][n][n][n][n][n][n][n]`. The number of values returned for each output variable is indicated in the `output_length` element of the `scf_data` structure. The `output_length` element is an array of values, with one value being returned per output variable. Indexing into this array starts at position zero. The user is referred to the EXAMPLE section for a coding example which exemplifies data retrieval for each output variable that is returned by this module.

The amount of time that is processed for each iteration of the algorithm is specified in the call to the `scf_sample_rate` routine. The user must call the `scf_sample_rate` routine once per program before the `scf_output_data` routine is called; otherwise, an error code is returned by this module. If the user specifies that the SCF software is to determine the accumulation rate and if the sample rate for one of the input variables changes while the algorithm is being executed, the software will continue to acquire data for the current accumulation period. If the sample rate changed such that the source is returning data at a rate faster than the current accumulation period, the accumulation period will be re-set at the next iteration of the algorithm.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- `scf_open` 3R
- `scf_sample_rate` 3R
- `create_scf_data_structure` 3R
- `scf_version_number` 3R
- `SCF_codes` 3H
- `libbase_SCF` 3H
- `scf_data` 3S

**BUGS**

None
EXAMPLES

Execute the algorithm defined in the SCF file TMMO_EXAMPLE one time. The data is returned in the `scf_data` structure referenced by the pointer `scf_data_ptr`. Print out the values for the output variables returned. The code assumes that the `scf_sample_rate` module has been called.

```c
#include "libbase_SCF.h"
#include "SCF_codes.h"

struct scf_data *SCF_DATA;
register SDDAS_LONG i;
SDDAS_FLOAT *dptr, *stop_loop;
SDDAS_USHORT scf_vnum;
SDDAS_SHORT status;
void *scf_data_ptr;

scf_version_number (&scf_vnum);
status = create_scf_data_structure (&scf_data_ptr);
if (status != ALL_OKAY)
{
  printf ("\n Error %d returned by create_scf_data_structure routine.\n", status);
  exit (-1);
}
SCF_DATA = (struct scf_data *) scf_data_ptr;

status = scf_output_data ("TMMO_EXAMPLE", scf_vnum, scf_data_ptr);
if (status != ALL_OKAY)
{
  printf ("\n Error %d returned by scf_output_data routine.\n", status);
  exit (-1);
}

/*  Print the output variables returned.                             */
for (i = 0; i < SCF_DATA->num_output; ++i)
{
  dptr = SCF_DATA->output_data + *(SCF_DATA->output_index + i);
  stop_loop = dptr + *(SCF_DATA->output_length + i);
  for (; dptr < stop_loop; ++dptr)
    printf ("\nOutput Variable %ld = %e", i, *dptr);
}
SCF_POSITION
function - positions the IDFS file pointers at the requested time in the files for all defined SCF input variables

SYNOPSIS
#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_position (SDDAS_CHAR *filename, SDDAS_USHORT scf_version,
SDDAS_SHORT btime_yr, SDDAS_SHORT btime_day,
SDDAS_LONG btime_sec, SDDAS_LONG btime_nano,
SDDAS_SHORT etime_yr, SDDAS_SHORT etime_day,
SDDAS_LONG etime_sec, SDDAS_LONG etime_nano)

ARGUMENTS
filename  - the name of the SCF file of interest
scf_version  - SCF identification number which allows for multiple
openings of the same SCF file
btime_yr  - the year at which algorithm execution is to commence
btime_day  - the day of year at which algorithm execution is to commence
btime_sec  - the time of day in seconds at which algorithm execution is to commence
btime_nano  - the time of day residual in nanoseconds
etime_yr  - the year at which algorithm execution is to terminate
etime_day  - the day of year at which algorithm execution is to terminate
etime_sec  - the time of day in seconds at which algorithm execution is to terminate
etime_nano  - the time of day residual in nanoseconds
scf_position  - routine status (see TABLE 1)

TABLE 1. Status Code Returned for SCF_POSITION

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_POS_ERROR</td>
<td>error returned from call to file_pos for specified input variables</td>
</tr>
<tr>
<td>CHK_TDATA_NOT_FOUND</td>
<td>an error was encountered when trying to access the structure that holds information pertaining to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>CHK_DATA_NOT_FOUND</td>
<td>an error was encountered when trying to access the structure that holds information pertaining to one of the IDFS data sets being processed</td>
</tr>
<tr>
<td>WRONG_DATA_STRUCTURE</td>
<td>incompatibility between IDFS data set and IDFS data structure used to hold the data being returned</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>Error codes returned by file_pos ()</td>
</tr>
</tbody>
</table>

the routine terminate successfully
Scf_position utilizes the IDFS read routine file_pos. For a complete listing of the error codes returned by this module, the user is referred to section 1R of the IDFS Programmers Manual.

DESCRIPTION

Scf_position positions all of the IDFS data sets at the requested start time. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. This routine uses the currently opened IDFS files that are associated with the defined input variables and sets the current data pointers to the data sample or sweep whose beginning time is closest to the requested start time. If all of the IDFS data sets cannot be positioned, an error code is returned. Before the first call to the scf_position routine can be made, a call to the routine scf_open with the identical filename and scf_version parameters must have been made to obtain a set of file descriptors for the appropriate vidf, header and data files.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

ERRORS

All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.

SEE ALSO

scf_open 3R
scf_version_number 3R
libbase_SCF 3H
SCF_codes 3H

BUGS

None

EXAMPLES

Position the IDFS data files associated with the input variables defined in the SCF file TMMO_EXAMPLE at the start of the time range specified.

#include "libbase_SCF.h"
#include "SCF_codes.h"
SDDAS_LONG btime_sec, btime_nsec, etime_sec, etime_nsec;
SDDAS_USHORT scf_vnum;
SDDAS_SHORT status, btime_yr, btime_day, etime_yr, etime_day;

btime_yr = 1992;
btime_day = 217;
btime_sec = 32340;
btime_nsec = 0;

etime_yr = 1992;
etime_day = 217;
etime_sec = 32342;
etime_nsec = 0;

scf_version_number (&scf_vnum);
status = scf_open ("TMMO_EXAMPLE", scf_vnum, btime_yr, btime_day, btime_sec,
              btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec);
if (status != ALL_OKAY)
{
    printf ("\nError %d returned by scf_open routine.\n", status);
    exit (-1);
}

status = scf_position ("TMMO_EXAMPLE", scf_vnum, btime_yr, btime_day,
              btime_sec, btime_nsec, etime_yr, etime_day, etime_sec, etime_nsec);
if (status != ALL_OKAY)
{
    printf("\nError %d returned by scf_position routine.\n", status);
    exit (-1);
}
SCF_SAMPLE_RATE

function - determines the amount of time to be processed for each iteration of the algorithm

SYNOPSIS

```c
#include "libbase_SCF.h"
#include "SCF_defs.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_sample_rate (SDDAS_CHAR *filename, 
   SDDAS_USHORT scf_version, SDDAS_CHAR accum_method, 
   SDDAS_DOUBLE time_value, SDDAS_CHAR rate_calc)
```

ARGUMENTS

- filename: the name of the SCF file of interest
- scf_version: SCF identification number which allows for multiple openings of the same SCF file
- accum_method: the scheme used to determine the amount of time to be processed for each iteration of the algorithm
  1. use the accumulation rate of the fastest input variable, as determined by the SCF software (SCF_DELTA_T)
  2. use the accumulation rate of the input variable specified in the `time_value` parameter (USE_INPUT_VAR)
  3. use the accumulation rate specified in the `time_value` parameter (USE_DELTA_T)
- time_value: the input variable number to use for the USE_INPUT_VAR accum_method or the time period to use as accumulation rate for the USE_DELTA_T accum_method
- rate_calc: the method to use to calculate the accumulation rate for the IDFS data sets utilized by the input variables
  1. use data accumulation values (SCF_MEASURE_TM)
  2. use data accumulation plus data latency values (SCF_MEASURE_LAT_TM)
- scf_sample_rate: routine status (see TABLE 1)

### TABLE 1. Status Codes Returned for SCF_SAMPLE_RATE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_SAMP_VECTOR_ACCUM</td>
<td>the accumulation scheme selected in the accum_method parameter is not a valid selection for non-scalar inputs variables</td>
</tr>
<tr>
<td>SCF_SAMP_POS</td>
<td>the IDFS data sets have not been positioned at the designated start time (user did not call scf_position for this combination)</td>
</tr>
</tbody>
</table>
### STATUS CODE | EXPLANATION OF STATUS
---|---
SCF_SAMP_SWP_MALLOC | no memory to hold the time period for each element of the sweep
SCF_SAMP_BAD_ACCUM | invalid value for the accum_method parameter
SCF_SAMP_BAD_LOCATE | an error was encountered when trying to access information pertinent to one of the IDFS data sets being processed
SCF_SAMP_BAD_RATE | invalid value for the rate_calc parameter
SCF_SAMP_BAD_INPUT_NUM | invalid value for the time_value parameter; bad input variable number
SCF_SAMP_VECTOR_SRC | a non-scalar input variable must be selected when the input variables are a combination of scalar and non-scalar (vector) data
SCF_FAST_BAD_LOCATE | an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed
CUR_TIME_NOT_FOUND | an error was encountered when trying to access the structure that holds information pertinent to one of the IDFS data sets being processed

| ALL_OKAY | the routine terminated successfully

### DESCRIPTION

The `scf_sample_rate` determines the amount of time to be processed for each iteration of the algorithm defined by the SCF. The SCF file of interest is referenced through the parameter `filename`. The `filename` parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. The user must call the `scf_position` routine before the `scf_sample_rate` module can be called. If the `scf_sample_rate` routine determines that the `scf_position` routine has not been called, an error code is returned to the user. In addition, this module must be called once per program prior to calling the `scf_output_data` routine; otherwise, the `scf_output_data` routine will return an error code.

The parameter `scf_version` allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the `scf_version_number` routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The amount of time to be processed for each iteration of the algorithm can be computed in one of three ways. The first method lets the user specify the amount of time to be processed. For this method, the parameter `accum_method` must be set to the value `USE_DELTA_T` and the parameter `time_value` must be set to the number of seconds to use as the accumulation period. Since this parameter is a double precision floating point value, fractions of a second can be specified. Since the amount of time is specified and not computed, the parameter `rate_calc` is not utilized, but it must still be specified. The user cannot select this method if any of the defined input variables are vector IDFS sources; an error code will be returned to the user if this condition is true.
The second method uses the data accumulation rate for a specific input variable as the amount of time to be processed for each iteration of the algorithm. For this method, the parameter `accum_method` must be set to the value `USE_INPUT_VAR` and the parameter `time_value` must be set to the input variable number. A check is made to ensure that the input variable number specified is a valid value. If the data associated with the defined input variables is a mixture of scalar and vector (1-D) data, the user must select a 1-D vector input variable to calculate the accumulation period. If the user specified a scalar input variable, an error code is returned to the user.

The third method should be selected when the SCF software is to determine the amount of time to be processed for each iteration of the algorithm. For this method, the parameter `accum_method` must be set to the value `SCF_DELTA_T`. The parameter `time_value` is not utilized but it must still be specified; a value of 0.0 is suggested. For this method, the software loops over all defined input variables to find the IDFS data set that has the fastest sample rate and that data set controls the rate of data accumulation for each iteration of the algorithm. If the data associated with the defined input variables is a mixture of scalar and vector (1-D) data, the SCF software will only use the vector data to determine the fastest sample rate.

For the second and third method, the `rate_calc` parameter defines the scheme that is to be used to calculate the sample rate for the IDFS data sets utilized by the input variables. For a scalar IDFS data set, if the parameter is set to `SCF_MEASURE_TM`, the sample rate will be determined by comparing the `data_accum` value found in the header record for the IDFS data sets. If the parameter is set to `SCF_MEASURE_LAT_TM`, the sample rate will be determined by combining the `data_accum` and `data_lat` values found in the header record for the IDFS data sets. The `data_accum` value is defined as the time over which the acquisition of a single datum occurs and the `data_lat` value is defined as the dead time between successive data acquisitions. The `data_accum` and the `data_lat` values together give the total time between successive accumulations.

For a vector IDFS data set, the interpretation of the `rate_calc` parameter is somewhat different. If the parameter is set to `SCF_MEASURE_TM`, the sample rate will be determined by comparing the time it takes to acquire the data for the sweep, which is defined as `(data_accum + data_lat)` times the number of samples in the sweep. If the parameter is set to `SCF_MEASURE_LAT_TM`, the sample rate will be determined by combining the time it takes to acquire the data for the sweep and the `swp_reset` value found in the header record for the IDFS data sets. The `swp_reset` value is defined as the dead time between successive columns of data, which is equivalent to any data latency which exists in going from the last step in one vector back to the initial step in the next vector.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.
SEE ALSO

scf_version_number  3R  
scf_output_data    3R  
scf_position      3R  
SCF_codes         3H  
SCF_defs          3H  
libbase_SCF       3H  

BUGS

None

EXAMPLES

Determine the fastest sample rate of all input variables defined in the SCF file TMMO_EXAMPLE. The following code segment assumes that scf_version_number and scf_position modules have been called.

```c
#include "libbase_SCF.h"
#include "SCF_defs.h"
#include "SCF_codes.h"

SDDAS_USHORT scf_vnum;
SDDAS_DOUBLE time_value;
SDDAS_SHORT status;

time_value = 0.0;
status = scf_sample_rate ("TMMO_EXAMPLE", scf_vnum, SCF_DELTA_T,
                            time_value, SCF_MEASURE_LAT_TM);

if (status != ALL_OKAY)
    {
    printf ("\n Error %d returned by scf_sample_rate routine.\n", status);
    exit (-1);
    }
```
SCF_TERMINATE_SOURCES

function – returns the IDFS data key(s) for the input variable(s) that are no longer available for processing and caused the return of the status code SCF_TERMINATE from the scf_output_data routine

SYNOPSIS

```c
#include "libbase_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_terminate_sources (SDDAS_CHAR *filename,
                                    SDDAS_USHORT scf_version,
                                    SDDAS_LONG *num_sources,
                                    SDDAS_ULONG **idfs_keys);
```

ARGUMENTS

- **filename** - the name of the SCF file of interest
- **scf_version** - SCF identification number which allows for multiple openings of the same SCF file
- **num_sources** - the number of elements defined for the `idfs_keys` array
- **idfs_keys** - pointer to the array of data keys for the IDFS data sources that are no longer available for processing (duplicates are removed)
- **scf_terminate_sources** - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_TERMINATE_SOURCES

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

The `scf_terminate_sources` function identifies the IDFS data source(s) that triggered the return value of SCF_TERMINATE from the scf_output_data routine. This return code indicates that some of the IDFS data sources are no longer available for processing for the time interval being executed. In other words, all of the data for the time step being processed by the SCF algorithm is not available online. The SCF file of interest is referenced through the parameter `filename`. The `filename` parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length.

The parameter `scf_version` allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the
The scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The data key(s) for the IDFS data source(s) that are no longer available for processing are returned in the idfs_keys array and the number of elements contained in the idfs_keys array is returned in num_sources. The memory for the idfs_keys array is allocated by this module and should be freed by the calling module once the information has been processed. The pointer value for the idfs_keys argument should be checked for a NULL value, which indicates that the memory allocation attempt failed.

**ERRORS**

All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- scf_version_number 3R
- scf_output_data 3R
- scf_open 3R
- SCF_codes 3H
- libbase_SCF 3H

**BUGS**

None

**EXAMPLES**

Execute the algorithm defined in the SCF file TMMO_EXAMPLE one time by calling the module scf_output_data. If the status code SCF_TERMINATE is returned, print out the data key(s) for the IDFS data source(s) that are no longer available for processing.

```c
#include "libbase_SCF.h"
#include "SCF_codes.h"

register SDDAS_LONG i;
SDDAS_LONG num_sources;
SDDAS_ULONG *data_keys;
SDDAS_USHORT scf_vnum;
SDDAS_SHORT status;
void *scf_data_ptr;

scf_version_number (&scf_vnum);
```
status = create_scf_data_structure (&scf_data_ptr);
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by create_scf_data_structure routine.\n", status);
    exit (-1);
}

status = scf_output_data ("TMMO_EXAMPLE", scf_vnum, scf_data_ptr);
if (status != ALL_OKAY && status != SCF_TERMINATE)
{
    printf("\n Error %d returned by scf_output_data routine.\n", status);
    exit (-1);
}

if (status == SCF_TERMINATE)
{
    status = scf_terminate_sources ("TMMO_EXAMPLE", scf_vnum, &num_sources,
&data_keys);
    if (status != ALL_OKAY)
    {
        printf("\n Error %d returned by scf_terminate_sources routine.\n", status);
        exit (-1);
    }

    if (*data_keys != NULL)
        for (i = 0; i < num_sources; ++i)
            printf("\nData Key[%ld] = %ld", i, data_keys[i]);

    free (data_keys);
}
SCF_VERSION_NUMBER
function - returns a unique SCF identification number

SYNOPSIS
#include "libbase_SCF.h"

void scf_version_number (SDDAS_USHORT *scf_version)

ARGUMENTS
    scf_version - SCF identification number which allows for multiple openings of the
                  same SCF file for concurrent algorithm execution

DESCRIPTION
Scf_version_number returns a unique SCF identification number that is to be used as a
parameter to the other SCF routines. This parameter allows multiple file openings for an
SCF file. In most cases, the user may open many different SCF files, opening each SCF file
once. In this case, the user may pass the same SCF version number for each of the different
SCF files; that is, one SCF version number is sufficient. The user should call the
scf_version_number module to be guaranteed a unique SCF version number. For multiple
file openings of the same SCF file, the SCF version number must be unique and all file
manipulations performed by the SCF routines will use the file descriptors defined for the
SCF version number specified.

ERRORS
This routine returns no status or error codes.

BUGS
None

EXAMPLES
Retrieve a unique SCF version number to be used by the SCF routines.

#include "libbase_SCF.h"

SDDAS_USHORT vnum;

scf_version_number (&vnum);
SCF_ALGORITHM_START function - returns the start time and the accumulation period (delta-t) for the first iteration of the SCF algorithm

SYNOPSIS

#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_algorithm_start (SDDAS_CHAR *filename, SDDAS_USHORT scf_version, SDDAS_SHORT *start_year, SDDAS_SHORT *start_day, SDDAS_LONG *start_sec, SDDAS_LONG *start_nano, SDDAS_LONG *res_sec, SDDAS_LONG *res_nano)

ARGUMENTS

filename - the name of the SCF file of interest
scf_version - SCF identification number which allows for multiple openings of the same SCF file
start_year - the year time component for the first iteration of the SCF algorithm
start_day - the day of year time component for the first iteration of the SCF algorithm
start_sec - the time of day in seconds for the first iteration of the SCF algorithm
start_nano - the time of day residual in nanoseconds for the first iteration of the SCF algorithm
res_sec - the accumulation period (delta-t) in seconds
res_nano - the accumulation period residual in nanoseconds
scf_algorithm_start - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_ALGORITHM_START

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_ALG_START_NO_SAMPLE</td>
<td>the data accumulation rate has not been set (user did not call scf_sample_rate for this combination)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>the routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

Scf_algorithm_start returns the start time and the amount of time processed for the first iteration of the algorithm defined by the SCF. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. The user must call the scf_sample_rate routine before the scf_algorithm_start module can be called. If
the `scf_algorithm_start` routine determines that the `scf_sample_rate` routine has not been called, an error code is returned.

The parameter `scf_version` allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the `scf_version_number` routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- `scf_open` 3R
- `scf_version_number` 3R
- `scf_sample_rate` 3R
- `SCF_codes` 3H
- `libavg_SCF` 4H

**BUGS**

None

**EXAMPLES**

Determine the start time and the time accumulation period associated with the first iteration of the SCF file TMMO_EXAMPLE. The following code segment assumes that `scf_version_number` and `scf_sample_rate` modules have been called.

```c
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_USHORT scf_vnum;
SDDAS_SHORT status, base_yr, base_day;
SDDAS_LONG base_sec, base_nano, res_sec, res_nano;

status = scf_algorithm_start ("TMMO_EXAMPLE", scf_vnum, &base_yr,
                          &base_day, &base_sec, &base_nano, &res_sec, &res_nano);
```
if (status != ALL_OKAY)
{
    printf("\n Error %d returned by scf_algorithm_start routine.\n", status);
    exit (-1);
}
SCF_BIN_INFO
function - specifies how SCF data is to be binned for time or sample averaging

SYNOPSIS
#include "libavg_SCF.h"
#include "SCF_codes.h"
#include "user_defs.h"

SDDAS_SHORT scf_bin_info (SDDAS_CHAR *filename,
                        SDDAS_USHORT scf_version, SDDAS_LONG output_var,
                        SDDAS_CHAR swp_type, SDDAS_FLOAT start,
                        SDDAS_FLOAT stop, SDDAS_LONG num_bins,
                        SDDAS_CHAR swp_fmt, SDDAS_LONG center_var,
                        SDDAS_LONG band_var, SDDAS_LONG upper_band_var,
                        SDDAS_CHAR var_fmt, SDDAS_CHAR input_fmt)

ARGUMENTS
filename - the name of the SCF file of interest
scf_version - SCF identification number which allows for multiple
openings of the same SCF file
output_var - output variable identification number (numbering starts at
zero)
swp_type - the format used to determine the number of data bins and data
storage
            1 - the number of bins used is equal to the
               number of values returned for the
               specified output variable
               (FIXED_SWEEP)
            2 - user specifies the number of bins
               (VARIABLE_SWEEP)
start - the center value associated with the first bin for variable
sweep processing
stop - the center value associated with the last bin for variable sweep
processing
num_bins - the number of bins to create for variable sweep processing
swp_fmt - the spacing for the bins
            1 - use linear spacing (LIN_SPACING)
            2 - use logarithmic spacing
               (LOG_SPACING)
            3 - use variable width spacing
               (VARIABLE_SPACING)
center_var - the output variable which holds the center values to be used
for the bins (the dependent variable); a value of -1 means no
output variable specified since numbering starts at zero
**band_var** - the output variable which holds the widths of the bins (used to create the band width values); a value of -1 means no output variable specified since numbering starts at zero.

**upper_band_var** - the output variable which holds the upper edges of the scan bins when **var_fmt** is set to ‘A’ or ‘a’; a value of -1 means no output variable specified since numbering starts at zero.

**var_fmt** - the format flag for variable width spacing

- **L or l** - the center bin values are used as the lower edge of the band width values
- **C or c** - the center bin values are used as the midpoints of the band width values
- **U or u** - the center bin values are used as the upper edge of the band width values
- **E or e** - the center bin values are used as the lower edge of the band width values and the scan widths specified are the actual upper edge of the band width values, not delta values
- **A or a** - the actual center, lower edge band width and upper edge band width values are provided (no computations off the center values are necessary).

**input_fmt** - the storage scheme for the binning of the SCF data for variable sweep processing

- **1** - data is placed in the bin which encompasses the value for the dependent variable associated with the data (POINT_STORAGE)
- **2** - data is placed in all bins which fully or partially contain the range for the dependent variable associated with the data (BAND_STORAGE)

**scf_bin_info** - routine status (see TABLE 1)

---

**TABLE 1. Status Codes Returned for SCF_BIN_INFO**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call <code>scf_open</code> for this combination)</td>
</tr>
<tr>
<td>SCF_BIN_BAD_SWP_FMT</td>
<td>invalid value specified for the <code>swp_fmt</code> argument</td>
</tr>
<tr>
<td>SCF_BIN_BAD_FMT</td>
<td>VARIABLE_SPACING can only be requested in conjunction with <code>FIXED_SWEEP</code> processing</td>
</tr>
<tr>
<td>SCF_BIN.BAD_OVAR_NUM</td>
<td>invalid value specified for the <code>output_var</code> argument</td>
</tr>
<tr>
<td>SCF_BIN.BAD_CNUM</td>
<td>no memory for data binning information</td>
</tr>
<tr>
<td>SCF_BIN.BAD_BNUM</td>
<td>invalid value specified for the <code>band_var</code> argument</td>
</tr>
<tr>
<td>SCF_BIN.BAD_VFMT</td>
<td>bad format character specified in <code>var_fmt</code> argument</td>
</tr>
</tbody>
</table>
**DESCRIPTION**

Scf_bin_info defines the size and the spacing of the data buffers that will be filled by the scf_time_average or scf_sample_average routine. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. Scf_bin_info must be called once for each output variable that is to be returned by the scf_time_average or scf_sample_average routine. This is necessary since the output variables returned by an SCF do not have to be homogeneous; that is, the data can be a mixture of scalar and multi-dimensional data. The first call to the scf_bin_info module for the output variable specified will be used to generate the binning information. All subsequent calls with the identical filename, scf_version and output_var parameters will be ignored. This module must be called prior to calling the scf_time_average / scf_sample_average routine.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

There are two formats that can be used to bin the data, FIXED_SWEEP and VARIABLE_SWEEP. With a FIXED_SWEEP format, the bins are set up according to the definition of the output variable (output_var) in the named SCF file. The number of bins used is equal to the number of values returned for the output variable; therefore, the value for the num_bins parameter is ignored. The data is always stored into the bins element by element, starting with element zero and terminating with the last element returned. When specifying a FIXED_SWEEP format, the values for the parameters start, stop and input_fmt are ignored. If the output variable specified is a scalar quantity or a variable with a dimension greater than or equal to 2-D, the scf_bin_info module will default to the FIXED_SWEEP format with linear spaced bins with no output variable specified for the center_var parameter, regardless of the setting of the parameters. The variable width spacing option (VARIABLE_SPACING) for the swp_fmt parameter is applicable only for the FIXED_SWEEP format. If the user tries to specify this option for the VARIABLE_SWEEP format, an error code is returned.
If the user selects a VARIABLE_SWEEP format, the user must specify the number of bins to create (num_bins), the center value associated with the first bin (start), the center value associated with the last bin (stop), the spacing of the bins (swp_fmt) and the scheme to use for storing the data (input_fmt). The data from an IDFS 1-D vector data set are taken as a function of a variable M. If the 1-D vector data is being returned as an output variable, the dependent variable M must also be returned. This output variable must be specified in the center_var parameter. If M is allowed to vary over the individual measurement period or if M actually represents a band width, then each element in the vector can be considered to have been accumulated with the interval M - δ1 to M + δ2. Vector data is binned by M. If the center variable M is located between the upper and lower edge values of a given bin, the data value is placed only in this bin. If the user selects the BAND STORAGE scheme, the data is placed in all bins which are fully or partially contained within the range M - δ1 to M + δ2. The data is multiplied by the percentage of the bin covered by the range before the data is placed into the bin.

The center and band width values for the bins are calculated once. The calculations are made after the first iteration of the SCF algorithm since data for output variables specified in the center_var, band_var, and upper_band_var parameters may be utilized. The algorithm used to create the center values is based in part upon the format selected (swp_type). If the VARIABLE_SWEEP format is selected, the center values are calculated using the start, stop and num_bins values. The difference between the start and stop value is computed and then divided by the number of bins requested. This value is added to the start value to calculate the next center value, with this process continuing until all centers have been calculated. The parameter swp_fmt specifies whether the centers are to be linearly or logarithmically spaced. The values for the center_var, band_var, and upper_band_var parameters are ignored.

If the FIXED_SPAWN format is selected, the center values will be computed in one of three ways. If variable width spacing (VARIABLE_SPACING) was selected for the swp_fmt parameter, the data returned by the output variable specified in the center_var parameter are used as the center values for the bins. If the swp_fmt parameter is set to LIN_SPACING or LOG_SPACING and if the center_var parameter specifies an output variable, the first and last data value for the output variable are used as the start and stop values and the computation is the same as described above for VARIABLE_SPAWN. If no output variable is specified for the center_var parameter (value set to -1), the center values are created, with values from zero to the number of bins requested minus one.

The algorithm used to compute the band width values for the bins is dependent upon the swp_fmt parameter. Linear spacing defines a scheme where the lower (upper) edge of the band is determined by subtracting (adding) one-half of the difference between two successive center values from (to) the center value. The same algorithm is used for log spacing, with the log of the center values being utilized. Variable width spacing defines a scheme where the data returned by the output variable specified in the band_var parameter are used as correction values that are to be applied to the center values in order to calculate the band width values. The variable var_fmt specifies how the correction values are to be
applied. If the var_fmt value is 'L' or 'l', the lower edge of the band is set to the center value and the upper edge of the band is calculated by adding the correction value to the center value. If the var_fmt value is 'C' or 'c', the lower edge of the band is calculated by subtracting one-half of the correction value from the center value and the upper edge of the band is calculated by adding one-half of the correction value to the center value. If the var_fmt value is 'U' or 'u', the lower edge of the band is calculated by subtracting the correction value from the center value and the upper edge of the band is set to the center value. If the var_fmt value is 'E' or 'e', the lower edge of the band is set to the center value and the upper edge of the band is set to the correction value; therefore, the correction value is not really a delta value, it is the actual value to be used as the upper edge of the band. If this format is selected, please take note that the center values and the lower edge values will be identical. If the var_fmt value is 'A' or 'a', there is no need to perform a computation using the center values in order to derive the lower and upper edges of the band. The “actual” values for the centers, lower edges and upper edges of the scan band are returned by the output variables specified in the parameters center_var, band_var, and upper_band_var, respectively. The user is referred to the scf_output_center_and_band write-up for more information concerning center and band width values.

ERRORS
All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.

SEE ALSO
scf_open 3R
scf_version_number 3R
scf_time_average 4R
scf_sample_average 4R
scf_output_center_and_band 4R
user_defs 1H
SCF_codes 3H
libavg_SCF 4H

BUGS
None

EXAMPLES
Create the data bins using a FIXED_SWEEP, linear spaced binning scheme for output variable zero defined in the SCF file TMMO_EXAMPLE. The following code segment assumes that the scf_version_number module has been called to set the scf_vnum parameter.

#include "SCF_codes.h"
#include "user_defs.h"
#include "libavg_SCF.h"
SDDAS_USHORT scf_vnum;
SDDAS_LONG dependent_var, output_var;
SDDAS_SHORT status;

output_var = 0;
dependent_var = -1;
status = scf_bin_info ("TMMO_EXAMPLE", scf_vnum, output_var,
FIXED_SWEEP, 0.0, 0.0, 1, LIN_SPACING, dependent_var,
dependent_var, dependent_var, \0', POINT_STORAGE);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by scf_bin_info routine.\n", status);
    exit (-1);
}

Create sixteen bins, starting at 5ev, stopping at 155ev, with log spacing and the data is to be
stored using BAND STORAGE for output variable zero.

#include "SCF_codes.h"
#include "libavg_SCF.h"
#include "user_defs.h"

SDDAS_USHORT scf_vnum;
SDDAS_LONG dependent_var, output_var;
SDDAS_SHORT status;

output_var = 0;
dependent_var = -1;
status = scf_bin_info ("TMMO_EXAMPLE", scf_vnum, output_var,
VARIABLE_SWEEP, 5.0, 155.0, 16, LOG_SPACING,
dependent_var, dependent_var, dependent_var, \0',
BAND_STORAGE);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by scf_bin_info routine.\n", status);
    exit (-1);
}

Create data bins using a FIXED_SWEEP/variable width spacing binning scheme for output
variable zero defined in the SCF file TMMO_EXAMPLE. The center values are returned
in output variable two and the band correction values are returned in output variable four.

#include "libavg_SCF.h"
#include "SCF_codes.h"
#include "user_defs.h"
SDDAS_USHORT scf_vnum;
SDDAS_SHORT status;
SDDAS_LONG center_var, band_var, upper_band_var, output_var;

center_var = 2;
band_var = 4;
upper_band_var = -1;
output_var = 0;

status = scf_bin_info ("TMMO_EXAMPLE", scf_vnum, output_var,
FIXED_SWEEP, 0.0, 0.0, 1, VARIABLE_SPACING, center_var,
band_var, upper_band_var, 'L', POINT_STORAGE);
if (status != ALL_OKAY)
{
printf ("\n Error %d returned by scf_bin_info routine.\n", status);
exit (-1);
}
**SCF_OUTPUT_CENTER_AND_BANDS**

function - returns the center and band width values associated with the data bins for the specified output variable

**SYNOPSIS**

```c
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_output_center_and_bands (SDDAS_CHAR *filename,
                                        SDDAS_USHORT scf_version, SDDAS_LONG output_var,
                                        SDDAS_FLOAT **center_ptr, SDDAS_FLOAT **low_ptr,
                                        SDDAS_FLOAT **high_ptr, SDDAS_LONG *num_bands)
```

**ARGUMENTS**

- **filename** - the name of the SCF file of interest
- **scf_version** - SCF identification number which allows for multiple openings of the same SCF file
- **output_var** - output variable identification number (numbering starts at zero)
- **center_ptr** - pointer to the location that holds the center bin values
- **low_ptr** - pointer to the location that holds the lower bands for non-contiguous bands or all band widths for contiguous bands
- **high_ptr** - pointer to the location that holds the upper bands for non-contiguous bands
- **num_bands** - the number of values returned
- **scf_output_center_and_bands** - routine status (see TABLE 1)

**TABLE 1.** Status Codes Returned for **SCF_OUTPUT_CENTER_AND_BANDS**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call <code>scf_open</code> for this combination)</td>
</tr>
<tr>
<td>SCF_OCENTER_OVAR_NUM</td>
<td>invalid value specified for the <code>output_var</code> argument</td>
</tr>
<tr>
<td>SCF_OCENTER_NO_AVG</td>
<td>the data has not been returned for the output variable (user did not call <code>scf_time_average</code> or <code>scf_sample_average</code>)</td>
</tr>
<tr>
<td>SCF_OCENTER_SELECT_MISSING</td>
<td>the output variable was not selected for processing (user did not call <code>scf_output_select</code> for this output variable)</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

`scf_output_center_and_bands` returns the center and band width values for bins that were created for the output variable `output_var` using the information specified by the call to the `scf_bin_info` routine. These center and band width values are used by the `scf_time_average` / `scf_sample_average` module when storing the data into the data bins for VARIABLE_SWEEP processing (refer to the explanation in the `scf_bin_info` write-up).
This module must be called after a call to the `scf_time_average` / `scf_sample_average` module has been made; otherwise, an error code will be returned. The SCF file of interest is referenced through the parameter `filename`. The `filename` parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length.

The parameter `scf_version` allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the `scf_version_number` routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The contents of the memory locations returned by this module should NOT be altered since the calculated center/band width values are used by the `scf_time_average` / `scf_sample_average` routine when processing the data. If the returned values need to be modified, for example, to take the log of the values, the user should allocate space to hold the values, copy the values into this space and modify the values there.

The module returns two possible pointers for the location(s) that hold the lower and upper band width values. In the case where the bands are non-contiguous, both the `low_ptr` and `high_ptr` will reference memory locations that hold the band width values. In the case where the bands are contiguous, there is no need to hold separate upper and lower values – the upper limit of the current band is the lower limit of the next band. In this case, one extra memory location is allocated, the `high_ptr` pointer is set to nil or 0 and `low_ptr` is set to reference the location that holds the band width values.

**ERRORS**

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- `scf_open` 3R
- `scf_version_number` 3R
- `scf_time_average` 4R
- `scf_sample_average` 4R
- `scf_bin_info` 4R
- `scf_output_select` 4R
- `SCF_codes` 3H
- `libavg_SCF` 4H
BUGS
None

EXAMPLES
Retrieve the center and band width values for the data bins created for output variable zero which is defined in the SCF file TMMO_EXAMPLE. The following code segment assumes that the scf_version_number module has been called to set the scf_vnum parameter.

```
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_USHORT scf_vnum;
register SDDAS_LONG bin;
SDDAS_LONG num_bins, output_var;
SDDAS_FLOAT *center_bin, *bin_low, *bin_high;
SDDAS_SHORT status;

output_var = 0;
status = scf_output_center_and_bands ("TMMO_EXAMPLE", scf_vnum,
                                      output_var, &center_bin, &bin_low,
                                      &bin_high, &num_bins);

if (status != ALL_OKAY)
{
    printf ("\n Error %d from scf_output_center_and_bands routine.\n", status);
    exit (-1);
}

/* Bands are contiguous? */

if (*bin_high == NULL)
    for (bin = 0; bin < num_bins; ++bin)
        printf ("\n low = %f high = %f", *(bin_low + bin), *(bin_low + bin + 1));

/* Bands are non-contiguous. */

else
    for (bin = 0; bin < num_bins; ++bin)
        printf ("\n low = %f high = %f", *(bin_low + bin), *(bin_high + bin));
```
SCF_OUTPUT_DATA_INDEX
function - returns index values to access the data returned by the scf_time_average / scf_sample_average modules

SYNOPSIS
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_output_data_index (SDDAS_CHAR *filename, 
SDDAS_USHORT scf_version, SDDAS_LONG output_var, 
SDDAS_FLOAT min, SDDAS_FLOAT max, 
SDDAS_LONG dependent_var, SDDAS_LONG *num_select, 
SDDAS_LONG *output_ind, SDDAS_ULONG *buf_zero_loc)

ARGUMENTS
filename   - the name of the SCF file of interest
scf_version   - SCF identification number which allows for multiple 
openings of the same SCF file
output_var   - output variable identification number (numbering 
starts at zero)
min    - the lower cutoff value for data that are to be put into 
the data buffers
max    - the upper cutoff value for data that are to be put into 
the data buffers
dependent_var   - the output variable whose data is to be used for the 
dependent variable for 1-D vector output (numbering 
starts at zero); a value of -1 should be used when 
output_var represents a scalar output variable
num_select   - the number of data sets returned for the output 
variable in question
output_ind   - an index value that is returned in order to access the 
correct sub-buffer (data set) returned for the output 
variable in question
buf_zero_loc   - an index value that is used to get to the beginning of 
all data returned for the output variable in question
scf_output_data_index - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_OUTPUT_DATA_INDEX

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_OINDEX_OVAR_NUM</td>
<td>invalid value specified for the output_var argument</td>
</tr>
<tr>
<td>SCF_OINDEX_NO_AVG</td>
<td>the data has not been returned for the output variable (user did not call scf_time_average or scf_sample_average)</td>
</tr>
<tr>
<td>SCF_OINDEX_NO_OUTPUT</td>
<td>the output variable was not selected for processing (user did not call scf_output_select for this output variable)</td>
</tr>
<tr>
<td>STATUS CODE</td>
<td>EXPLANATION OF STATUS</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SCF_OINDEX_NO_MATCH</td>
<td>the data cutoff/dependent_var combination requested was not found amongst the defined combinations for this output variable</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

DESCRIPTION

`scf_output_data_index` returns index values that are used to access the data buffers returned by the `scf_time_average` / `scf_sample_average` routines. The SCF file of interest is referenced through the parameter `filename`. The `filename` parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length.

The `scf_time_average` and `scf_sample_average` routines return a single pointer to the data array which holds the data for all output variables that were processed. For each output variable specified through the `scf_output_select` module, the `scf_time_average` routine returns `NUM_BUFFERS` working buffers, with `N` many sub-buffers, where `N` reflects the number of data cutoff/dependent_var combinations defined for the selected output variable. The `scf_sample_average` routine returns one working buffer, with `N` many sub-buffers for the selected output variable. In both scenarios, the value for `N` may vary from output variable to output variable. The index values returned by this module are used to access the data for a specific output variable, with specific data cutoff/dependent_var values.

The parameter `scf_version` allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the `scf_version_number` routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The user may elect to call the `scf_output_data_index` routine every time a return from the `scf_time_average` / `scf_sample_average` routine is made or may call the `scf_output_data_index` routine once for each output variable, data cutoff/dependent_var combination requested and save the index values for later usage. In either case, the call to the `scf_output_data_index` routine must be made after a call to the `scf_time_average` / `scf_sample_average` routine has been made; otherwise, an error code is returned.

ERRORS

All errors within this routine are returned through the status variable. The include file `SCF_codes.h`, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in section 3H of the IDFS Programmers Manual.
SEE ALSO

scf_open 3R
scf_version_number 3R
scf_time_average 4R
scf_sample_average 4R
scf_output_select 4R
SCF_codes 3H
libavg_SCF 4H

BUGS

None

EXAMPLES

Retrieve the index values to access data that is returned for scalar output variable zero which is defined in the SCF file TMMO_EXAMPLE. The data which uses a data cutoff range of 10.0 to 25.0 is to be accessed. The following code segment assumes that scf_version_number module has been called to set the scf_vnum parameter.

```c
#include "SCF_codes.h"
#include "libavg_SCF.h"

SDDAS_USHORT scf_vnum;
SDDAS_ULONG buf_zero_loc;
SDDAS_FLOAT data_min, data_max;
SDDAS_LONG output_var, dependent_var, num_select, output_ind;
SDDAS_SHORT status;

data_min = 10.0;
data_max = 25.0;
output_var = 0;
dependent_var = -1;

status = scf_output_data_index("TMMO_EXAMPLE", scf_vnum, output_var,
data_min, data_max, dependent_var, &num_select,
&output_ind, &buf_zero_loc);

if (status != ALL_OKAY)
{
    printf("\n Error %d returned by scf_output_data_index routine.\n", status);
    exit (-1);
}
```
SCF_OUTPUT_SELECT

function - specifies which output variable(s) are to be returned by the SCF time / sample average routines

SYNOPSIS
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_output_select (SDDAS_CHAR *filename,
                          SDDAS_USHORT scf_version, SDDAS_LONG output_var,
                          SDDAS_FLOAT min, SDDAS_FLOAT max,
                          SDDAS_LONG dependent_var)

ARGUMENTS
filename   - the name of the SCF file of interest
scf_version   - SCF identification number which allows for multiple
                openings of the same SCF file
output_var   - output variable identification number (numbering
                starts at zero)
min    - the lower cutoff value for data that are to be put into
        the data buffers
max    - the upper cutoff value for data that are to be put into
        the data buffers
dependent_var   - the output variable whose data is to be used for the
                    dependent variable for 1-D vector output (numbering
                    starts at zero); a value of -1 should be used when
                    output_var represents a scalar output variable
scf_output_select  - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_OUTPUT_SELECT

<table>
<thead>
<tr>
<th>STATUS_CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_SELECT_OVAR_NUM</td>
<td>invalid value specified for the output_var argument</td>
</tr>
<tr>
<td>SCF_SELECT_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call scf_bin_info for this combination)</td>
</tr>
<tr>
<td>SCF_SELECT_MALLOC</td>
<td>no memory for structures which hold output variable selection</td>
</tr>
<tr>
<td>SCF_SELECT_DEF_MALLOC</td>
<td>no memory for min/max/dependent_var values for the output variable being processed</td>
</tr>
<tr>
<td>SCF_SELECT_DEF_REALLOC</td>
<td>no memory for expansion of the min/max/dependent_var values for the output variable being processed</td>
</tr>
<tr>
<td>SCF_SELECT_DVAR_NUM</td>
<td>invalid value specified for the dependent_var argument</td>
</tr>
<tr>
<td>SCF_SELECT_DVAR_LENGTH</td>
<td>the length of the dependent_var output variable is not the same as the length of the output_var output variable</td>
</tr>
<tr>
<td>SCF_SELECT_BAND_MALLOC</td>
<td>no memory for the band width values created from data for the dependent_var output variable</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>
DESCRIPTION

Scf_output_select specifies which output variable(s) are to be returned by the scf_time_average / scf_sample_average routines. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. Scf_output_select must be called at least once for each output variable that is to be time or sample averaged and must be called before the scf_time_average or scf_sample_average routine can be called. In addition, the routine scf_bin_info must be called for the same output variable as that specified in output_var before this module can be called; otherwise, an error code is returned by the scf_output_select routine.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The output variable dependent_var is utilized when the data bins are processed using a VARIABLE_SWEEP format. When using the VARIABLE_SWEEP format, the data for the output variable output_var is assumed to be taken as a function of a variable M, which is termed the dependent variable. The dependent variable M must also be returned as an output variable by the SCF. This output variable must be specified in the dependent_var parameter. The variable output_var may be associated with multiple dependent variables; that is, the same data can be associated with many dependent parameters as long as those dependent parameters are being returned as output variables in the SCF. When this is the case, multiple calls to the scf_output_select routine must be made, using the same output_var value with different dependent_var values. The user is referred to the scf_bin_info write-up for more information concerning data bins and storage.

ERRORS

All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.

SEE ALSO

  scf_open    3R
  scf_version_number    3R
  scf_bin_info    4R
  scf_time_average    4R
  scf_sample_average    4R
  SCF_codes    3H
EXAMPLES
Specify cutoff values for output variables zero and one, which are defined in the SCF file TMMO_EXAMPLE. Output variable zero is scalar and output variable one is a 1-D vector, with the data for the dependent variable being returned in output variable two. The following code segment assumes that scf_version_number module has been called to set the scf_vnum parameter.

```c
#include "libavg_SCF.h"
#include "user_defs.h"
#include "SCF_codes.h"

SDDAS_USHORT scf_vnum;
SDDAS_LONG output_var, dependent_var;
SDDAS_FLOAT data_min, data_max;
SDDAS_SHORT status;

output_var = 0;
dependent_var = -1;
data_min = 10.0;
data_max = 25.0;
ret_val = scf_output_select ("TMMO_EXAMPLE", scf_vnum, output_var, data_min,
data_max, dependent_var);

if (status != ALL_OKAY)
{
    printf ("\n Error %d from scf_output_select routine.\n", status);
    exit (-1);
}

output_var = 1;
dependent_var = 2;
data_min = VALID_MIN;
data_max = VALID_MAX;
ret_val = scf_output_select ("TMMO_EXAMPLE", scf_vnum, output_var, data_min,
data_max, dependent_var);

if (status != ALL_OKAY)
{
    printf ("\n Error %d from scf_output_select routine.\n", status);
    exit (-1);
}
```

BUGS
None
SCF_SAMPLE_AVERAGE

function - returns sample-averaged data buffers for selected SCF output variables

SYNOPSIS

#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_SHORT scf_sample_average (SDDAS_CHAR *filename,
SDDAS_USHORT scf_version, void *scf_data_ptr,
SDDAS_LONG num_iterations, SDDAS_FLOAT **ret_data,
SDDAS_FLOAT **ret_frac, SDDAS_CHAR **bin_stat,
SDDAS_SHORT *stime_yr, SDDAS_SHORT *stime_day,
SDDAS_LONG *stime_sec, SDDAS_LONG *stime_nano,
SDDAS_SHORT *etime_yr, SDDAS_SHORT *etime_day,
SDDAS_LONG *etime_sec, SDDAS_LONG *etime_nano,
SDDAS_LONG *num_output, SDDAS_LONG **output_var,
SDDAS_LONG **output_size)

ARGUMENTS

filename       - the name of the SCF file of interest
scf_version    - SCF identification number which allows for multiple
                openings of the same SCF file
scf_data_ptr   - pointer to the scf_data structure that temporarily holds
                the data for all output variables that are returned by the
                SCF algorithm
num_iterations - the number of samples (iterations of the SCF
                algorithm) to average together
ret_data       - pointer to the data being returned (data for output
                variables that are processed)
ret_frac        - pointer to the normalization factors for the data being
                returned
bin_stat        - pointer to status flags which are associated with each
                data bin returned

0 - no data has been placed into the data bin being processed
1 - data has been placed into the data bin being processed

stime_yr      - the year value for the first iteration of the SCF
                algorithm
stime_day      - the day of year value for the first iteration of the SCF
                algorithm
stime_sec      - the time of day in seconds for the first iteration of the
                SCF algorithm
stime_nano     - the time of day residual in nanoseconds for the first
                iteration of the SCF algorithm
etime_yr - the year value for the last iteration of the SCF algorithm
etime_day - the day of year value for the last iteration of the SCF algorithm
etime_sec - the time of day in seconds for the last iteration of the SCF algorithm
etime_nano - the time of day residual in nanoseconds for the last iteration of the SCF algorithm
num_output - the number of output variables processed (number of elements in the output_var and output_size arrays)
output_var - an array which holds the output variable number(s) for which data is returned (numbering starts with zero)
output_size - an array which holds the number of data values returned in a data buffer for each output variable that is processed
scf_sample_average - routine status (see TABLE 1)

**TABLE 1. Status Codes Returned for SCF_SAMPLE_AVERAGE**

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has no memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_SAVG_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call scf_bin_info for this combination)</td>
</tr>
<tr>
<td>SCF_SAVG_SELECT_MISSING</td>
<td>no output variables have been selected for processing (user did not call scf_output_select for any output variable)</td>
</tr>
<tr>
<td>SCF_SAMPLE_WITH_TIME</td>
<td>the modules scf_sample_average and scf_time_average cannot be used interchangeably for the same filename, scf_version combination</td>
</tr>
<tr>
<td>SCF_AVG_STR_MALLOC</td>
<td>no memory for structure which hold information pertinent to the sample-averaged data</td>
</tr>
<tr>
<td>SCF_SINFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SCF_SDATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>SCF_CENTER_MALLOC</td>
<td>no memory for center bin values</td>
</tr>
<tr>
<td>SCF_BAND_MALLOC</td>
<td>no memory for bin band width values</td>
</tr>
<tr>
<td>SCF_TERMINATE</td>
<td>processing must stop due to data not being on-line</td>
</tr>
<tr>
<td>SCF_NO_CENTER_VALUES</td>
<td>no values are available to compute the center bin values for the specified center variable</td>
</tr>
<tr>
<td>SCF_BAD_START_STOP</td>
<td>the start / stop scan value is outside of the valid data range for the parameter specified as the scan parameter</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>routine terminated successfully</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

Scf_sample_average is the SCF sample-averaged data read routine, averaging iterations of the SCF algorithm for all selected output variables. The SCF file of interest is referenced through the parameter **filename**. The **filename** parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. The data that is returned is dictated by the output variables that are selected using the scf_output_select routine. If no output variables were selected for the SCF
filename/version combination, an error code is returned; otherwise, the number of output variables processed is returned in the **num_output** parameter, along with the output variable number(s) and the number of elements in the data buffers.

The parameter **scf_version** allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the **scf_version_number** routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The parameter **scf_data_ptr** is a pointer to the structure that is to hold all data pertinent to the SCF file being processed. The structure is created and the address to this structure is returned when a call to the **create_scf_data_structure** routine is made. The contents of this structure is described in section 3S of the IDFS Programmers Manual. Since the SCF file dictates the number of output variables and the dimensionality of these variables, the user should call the **create_scf_data_structure** routine once for each distinct SCF file being processed and this pointer should be passed in conjunction with the named SCF file when the output variable values are being retrieved.

The data is processed one iteration at a time. Once the requested number of iterations have been processed, the routine will return the data. If the requested number of iterations could not be processed due to data acquisition problems, the routine will return the data and the normalization factors will reflect the number of iterations processed so far. If more data is put online, the next call to the **scf_sample_average** routine will continue to accumulate data and will continue until the remaining iterations have been acquired.

There are N many subbuffers, where N reflects the number of data cutoff/dependent_var combinations defined for the selected output variable. The value for N may vary from output variable to output variable. The user must process the data contained within these buffers before the next call to the **scf_sample_average** routine is made since the module will clear out these buffers for re-use if the requested number of iterations were processed on the previous call. The data values must be normalized using the normalization factors returned along with the data. The user is advised to check the value or values in the **bin_stat** array. If all values are 0, no data was placed into the buffer. This can happen if the data is excluded based upon data cutoff values.

The size and spacing of the data buffers are either defined by the user or by elements contained within the SCF file. The user must call the **scf_bin_info** module for each output variable that is to be returned before calling the **scf_sample_average** routine in order to specify how the binning of the data is to occur. If the **scf_sample_average** routine determines that no binning scheme has been selected, an error code is returned to the user.
ERRORS
All errors within this routine are returned through the status variable. The include file SCF_codes.h, which includes all possible return values, should be included so that the mnemonics for the return codes can be referenced. The SCF_codes.h file is described in section 3H of the IDFS Programmers Manual.

SEE ALSO
scf_open 3R
scf_output_data 3R
create_scf_data_structure 3R
scf_version_number 3R
scf_time_average 4R
scf_output_select 4R
scf_bin_info 4R
SCF_codes 3H
libavg_SCF 4H

BUGS
None

EXAMPLES
Obtain data one iteration at a time for output variables selected from the SCF file TMMO_EXAMPLE. This code segment assumes all necessary subroutine calls have been made.

```c
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_USHORT scf_vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG stime_sec, stime_nano, end_time_sec, end_time_nano;
SDDAS_LONG *output_numbers, num_output, *output_size;
SDDAS_SHORT stime_yr, stime_day, end_time_yr, end_time_day, status;
SDDAS_CHAR *ret_bin;
void *scf_data_ptr;

ret_val = scf_sample_average ("TMMO_EXAMPLE", scf_vnum, scf_data_ptr, 1,
    &ret_data, &ret_frac, &ret_bin, &stime_yr,
    &stime_day, &stime_sec, &stime_nano, &end_time_yr,
    &end_time_day, &end_time_sec, &end_time_nano,
    &num_output, &output_numbers, &output_size);
if (status != ALL_OKAY)
{
    printf ("\nError %d returned by scf_sample_average routine.\n", status);
    exit (-1);
}
```
SCF_TIME_AVERAGE
function - returns time-averaged data buffers for selected SCF output variables

SYNOPSIS
#include "libavg_SCF.h"
#include "SCF_codes.h"
#include "user_defs.h"

SDDAS_SHORT scf_time_average (SDDAS_CHAR *filename, 
SDDAS_USHORT scf_version, void *scf_data_ptr, 
SDDAS_FLOAT **ret_data, SDDAS_FLOAT **ret_frac, 
SDDAS_CHAR **bin_stat, SDDAS_LONG **bpix, 
SDDAS_LONG **epix, SDDAS_CHAR **ret_stat, 
SDDAS_SHORT *stime_yr, SDDAS_SHORT *stime_day, 
SDDAS_LONG *stime_sec, SDDAS_LONG *stime_nano, 
SDDAS_SHORT *etime_yr, SDDAS_SHORT *etime_day, 
SDDAS_LONG *etime_sec, SDDAS_LONG *etime_nano, 
SDDAS_LONG *num_output, SDDAS_LONG **output_var, 
SDDAS_LONG **output_size)

ARGUMENTS
filename     - the name of the SCF file of interest
scf_version  - SCF identification number which allows for multiple 
               openings of the same SCF file
scf_data_ptr - pointer to the scf_data structure that temporarily holds the 
               data for all output variables that are returned by the SCF 
               algorithm
ret_data     - pointer to the data being returned (data for output variables 
               that are processed)
ret_frac     - pointer to the normalization factors for the data being returned
bin_stat     - pointer to status flags which are associated with each data bin returned
  0 - no data has been placed into the data bin being processed
  1 - data has been placed into the data bin being processed
bpix         - pointer to the starting pixel location for the data buffers returned
epix         - pointer to the ending pixel location for the data buffers returned
ret_stat     - pointer to the status of each of the data buffers being returned
  UNTOUCHED_BUFFER - no data has ever been placed into the buffer
  FREE_BUFFER     - no data has been placed into the buffer being processed (ready for re-
scf_time_average (4R)

PARTIAL_WORKING - data is being acquired into the buffer but is not ready for processing

BUFFER_READY - data has been acquired into the buffer and is ready for processing

stime_yr - the year value for the first iteration of the SCF algorithm
stime_day - the day of year value for the first iteration of the SCF algorithm
stime_sec - the time of day in seconds for the first iteration of the SCF algorithm
stime_nano - the time of day residual in nanoseconds for the first iteration of the SCF algorithm
etime_yr - the year value for the last iteration of the SCF algorithm
etime_day - the day of year value for the last iteration of the SCF algorithm
etime_sec - the time of day in seconds for the last iteration of the SCF algorithm
etime_nano - the time of day residual in nanoseconds for the last iteration of the SCF algorithm
num_output - the number of output variables processed (the number of elements in the output_var and output_size arrays)
output_var - an array which holds the output variable number(s) for which data is returned (numbering starts with zero)
output_size - an array which holds the number of data values returned in a data buffer for each output variable that is processed
scf_time_average - routine status (see TABLE 1)

TABLE 1. Status Codes Returned for SCF_TIME_AVERAGE

<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE_SCF_NOT_FOUND</td>
<td>the requested filename, scf_version combination has not memory allocated for processing (user did not call scf_open for this combination)</td>
</tr>
<tr>
<td>SCF_TAVG_NO_BASE_TIME</td>
<td>the time interval information has not been set (user did not call scf_time_reference for this combination)</td>
</tr>
<tr>
<td>SCF_TAVG_BIN_MISSING</td>
<td>the data binning information has not been allocated (user did not call scf_bin_info for this combination)</td>
</tr>
<tr>
<td>SCF_TAVG_SELECT_MISSING</td>
<td>no output variable have been selected for processing (user did not call scf_output_select for any output variable)</td>
</tr>
<tr>
<td>SCF_TIME_WITH_SAMPLE</td>
<td>the modules scf_time_average and scf_sample_average cannot be used interchangeably for the same filename, scf_version combination</td>
</tr>
<tr>
<td>SCF_NO_EMPTY_BUFFERS</td>
<td>no spare buffers for data accumulation</td>
</tr>
<tr>
<td>SCF_AVG_STR_MALLOC</td>
<td>no memory for structure which hold information pertinent to the time averaged data</td>
</tr>
<tr>
<td>SCF_TINFO_MALLOC</td>
<td>no memory for data buffer information</td>
</tr>
<tr>
<td>SCF_TDATA_MALLOC</td>
<td>no memory for data buffers</td>
</tr>
<tr>
<td>SCF_CENTER_MALLOC</td>
<td>no memory for center bin values</td>
</tr>
</tbody>
</table>

400  December 28, 2012
<table>
<thead>
<tr>
<th>STATUS CODE</th>
<th>EXPLANATION OF STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCF_BAND_MALLOC</td>
<td>no memory for bin band width values</td>
</tr>
<tr>
<td>SCF_TERMINATE</td>
<td>processing must stop due to data not being on-line</td>
</tr>
<tr>
<td>SCF_NO_CENTER_VALUES</td>
<td>no values are available to compute the center bin values for the specified center variable</td>
</tr>
<tr>
<td>SCF_BAD_START_STOP</td>
<td>the start / stop scan value is outside of the valid data range for the parameter specified as the scan parameter</td>
</tr>
<tr>
<td>ALL_OKAY</td>
<td>Error codes returned by scf_output_data ()</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

**Scf_time_average** is the SCF time-averaged data read routine, retrieving data for all selected output variables for the time duration being processed. The SCF file of interest is referenced through the parameter filename. The filename parameter includes the full pathname extension of the SCF file being referenced and must be less than 512 characters in length. The data that is returned is dictated by the output variables that are selected using the scf_output_select routine. If no output variables were selected for the SCF filename/version combination, an error code is returned; otherwise, the number of output variables processed is returned in the num_output parameter, along with the output variable number(s) and the number of elements in the data buffers.

The parameter scf_version allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the scf_version_number routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

The parameter scf_data_ptr is a pointer to the structure that is to hold all data pertinent to the SCF file being processed. The structure is created and the address to this structure is returned when a call to the create_scf_data_structure routine is made. The contents of this structure is described in section 3S of the IDFS Programmers Manual. Since the SCF file dictates the number of output variables and the dimensionality of these variables, the user should call the create_scf_data_structure routine once for each distinct SCF file being processed and this pointer should be passed in conjunction with the named SCF file when the output variable values are being retrieved.

This routine will process sequential iterations of the SCF algorithm, placing the data into buffers which hold data that is accumulated over a specified time interval. In doing so, multiple data samples may be averaged together in a single buffer. Once the time interval has been processed, the routine will return the data buffers and a status value for each buffer which indicates when the buffer is ready for the user to retrieve. The user must call the module scf_time_reference before the scf_time_average module can be called since the scf_time_reference module is used to specify the base time value and reference location.
and the time interval (delta) to use to accumulate the data. If the `scf_time_average` routine
determines that the `scf_time_reference` routine has not been called, an error code is
returned.

Along with the data being returned, there is a starting location and an ending location that is
returned for each of the data buffers. The user may use these values as references to the
base location specified in the call to the `scf_time_reference` routine. That is, given a base
time value, a time interval and a reference location, the `scf_time_average` routine will
return the location of SCF data with respect to time. The user may chose to ignore these
values or may use these locations to plot data along an axis that is scaled with respect to
time.

There are a constant number of data buffers that are used by the `scf_time_average` module.
This number is defined as `NUM_BUFFERS` in the `user_defs.h` file. This file is described
in section 1H of the IDFS Programmers Manual. These data buffers are utilized in a cyclic
nature, with buffer 0 being re-used once buffer `NUM_BUFFERS-1` has been filled. The
data buffers that are ready to be processed are flagged with the status value
`BUFFER_READY`. For each buffer, there are N many sub-buffers, where N reflects the
number of data cutoff/dependent_var combinations defined for the selected output variable.
The value for N may vary from output variable to output variable. The user must process
the data contained within these buffers before the next call to the `scf_time_average` routine
is made since the module will clear out these buffers for re-use. The data values must be
normalized using the normalization factors returned along with the data. Since the buffers
are cyclic, the user may wish to keep a variable indicating the last buffer number processed
so that the user can start at the time sample left off from the previous call to the
`scf_time_average` routine at the next call. It is important to note that there is one status flag
per data buffer that is used by all output variables. If no data was placed into the buffer due
to the data being out of the specified range, the result will be that a buffer is flagged as
`BUFFER_READY` but will not contain any data. The user is advised to check the value or
values in the `bin_stat` array. If all values are 0, no data was placed into the buffer.

The size and spacing of the data buffers are either defined by the user or by elements
contained within the SCF file. The user must call the `scf_bin_info` module for each output
variable that is to be returned before calling the `scf_time_average` routine in order to
specify how the binning of the data is to occur. If the `scf_time_average` routine determines
that no binning scheme has been selected, an error code is returned to the user.

**ERRORS**

All errors within this routine are returned through the status variable. The include file
`SCF_codes.h`, which includes all possible return values, should be included so that the
mnemonics for the return codes can be referenced. The `SCF_codes.h` file is described in
section 3H of the IDFS Programmers Manual.

**SEE ALSO**

- `scf_open` 3R
- `scf_version_number` 3R

402 December 28, 2012
EXEMPLARY

EXAMPLES
Obtain time-averaged data for output variables selected from the SCF file TMMO_EXAMPLE. This code segment assumes all necessary subroutine calls have been made.

```c
#include "libavg_SCF.h"
#include "SCF_codes.h"

SDDAS_USHORT scf_vnum;
SDDAS_FLOAT *ret_data, *ret_frac;
SDDAS_LONG stime_sec, stime_nano, end_time_sec, end_time_nano;
SDDAS_LONG *output_numbers, num_output, *output_size, *bpix, *epix;
SDDAS_SHORT stime_yr, stime_day, end_time_yr, end_time_day, status;
SDDAS_CHAR *ret_bin, *buf_stat;
void *scf_data_ptr;

status = scf_time_average ("TMMO_EXAMPLE", scf_vnum, scf_data_ptr,
                        &ret_data, &ret_frac, &ret_bin, &bpix, &epix, &buf_stat,
                        &stime_yr, &stime_day, &stime_sec, &stime_nano,
                        &end_time_yr, &end_time_day, &end_time_sec,
                        &end_time_nano, &num_output, &output_numbers,
                        &output_size);

if (status != ALL_OKAY)
{
    printf ("\n Error %d returned by scf_time_average routine.\n", status);
    exit (-1);
}
```
**SCF_TIME_REFERENCE**

function - sets the base reference time, reference location and time duration to be utilized by the **scf_time_average** module

**SYNOPSIS**

#include "libavg_SCF.h"

void scf_time_reference (SDDAS_USHORT scf_version, SDDAS_SHORT base_year, SDDAS_SHORT base_day, SDDAS_LONG base_sec, SDDAS_LONG base_nano, SDDAS_LONG base_pix, SDDAS_LONG res_sec, SDDAS_LONG res_nano)

**ARGUMENTS**

- **scf_version** - SCF identification number which allows for multiple openings of the same SCF file
- **base_year** - the year time component for the base reference time
- **base_day** - the day of year time component for the base reference time
- **base_sec** - the time of day in seconds for the base reference time
- **base_nano** - the time of day residual in nanoseconds for the base reference time
- **base_pix** - the reference point or location associated with the base reference time
- **res_sec** - the time duration (delta-t) in seconds
- **res_nano** - the time duration residual in nanoseconds

**DESCRIPTION**

**Scf_time_reference** sets the base reference time, the reference location and the time duration values to be used by the **scf_time_average** routine. This routine should be called once, after all calls to the **scf_open** routine have been made. If the base reference time or the time duration is not known, the user can make a call to the **scf_algorithm_start** module in order to retrieve the start time and the accumulation period (delta-t) for the first iteration of the SCF algorithm.

The parameter **scf_version** allows multiple file openings for the same SCF file. If the SCF file needs to be opened just once for processing, the same SCF version number should be passed to all SCF routines. However, for multiple file openings, the SCF version number should be unique and all file manipulations performed by the SCF routines will use the file descriptors defined for the SCF version number specified. The user should call the **scf_version_number** routine to retrieve a unique SCF version number instead of choosing this value themselves. The retrieval of multiple output values from a single SCF source does not constitute the need for multiple SCF version numbers; a single SCF version number will suffice.

**ERRORS**

This routine returns no status or error codes.
SEE ALSO

- scf_time_average 4R
- scf_algorithm_start 4R
- scf_open 3R
- scf_version_number 3R
- libavg_SCF 4H

BUGS

None

EXAMPLES

The base reference time to be utilized is 1992, day 23, time 00:25:36 which is equal to 1536 seconds. The resolution to be utilized is 1.500 seconds and the reference location is at zero. Assume that the variable scf_vnum has been set by a previous call to the scf_version_number routine.

```c
scf_time_reference (scf_vnum, 1992, 23, 1536, 0, 0, 1, 500000000);
```
LIBBASE_IDFS.H
function - contains prototypes for the basic set of IDFS data retrieval routines

SYNOPSIS
#include "libbase_idfs.h"

DESCRIPTION
The libbase_idfs.h include file contains the ANSI C prototypes for the basic set of IDFS data retrieval routines that return data one sample set or one spin at a time. These routines can be found in the 1R section of the IDFS Programmers Manual. This file should be included in the source code wherever an IDFS routine is called from to ensure that the correct number of arguments are used and to ensure that the argument types match.

SEE ALSO
libtrec_idfs 2H
**RET_CODES.H**

function - defines possible return values and associated mnemonics for the IDFS software

**SYNOPSIS**

```c
#include "ret_codes.h"
```

**DESCRIPTION**

The *ret_codes.h* include file holds all of the defined return values for the IDFS routines. All return values are associated with a mnemonic through a `define` statement. The user of the IDFS routines should include this file and use the mnemonics defined for the return values so that if these return values are changed in the future to some other value, code does not have to be modified.

**CONTENTS OF FILE**

```c
#define ALL_OKAY 1
#define EOF_STATUS 0
#define LOS_STATUS 9
#define NEXT_FILE_STATUS 12
#define READ_SPIN_TERMINATE 15
#define READ_SPIN_DATA_GAP 18
#define CENTER_CONVERSION 4
#define DREC_NO_SENSOR 2
#define DREC_EOF_NO_SENSOR 3
#define DREC_EOF_SENSOR 6
#define TENSOR_NO_SENSOR 2
#define TENSOR_EOF_NO_SENSOR 3
#define TENSOR_EOF_SENSOR 6
#define RESET_CSET_MALLOC -1
#define BUF_BIN_MALLOC -2
#define LOCATE_NOT_FOUND -3
#define LOCATE_PTR_MALLOC -4
#define LOCATE_EX_REALLOC -5
#define CP_BAD_FRAC -6
#define CP_BAD_TIMES -7
#define OPEN_PTR_MALLOC -8
#define OPEN_EX_REALLOC -9
#define RTIME_NO_HEADER -10
#define RTIME_NO_DATA -11
#define CUR_TIME_NOT_FOUND -12
#define ALL_FLAG_MALLOC -13
#define ALLOC_HDR_READ_ERROR -14
#define ALLOC_HDR_MALLOC -15
#define ALLOC_HDR_REALLOC -16
#define CAL_DATA_MALLOC -17
#define PARTIAL_READ -18
#define SEL_SEN_NOT_FOUND -19
```
```c
#define TIME_OFF_MALLOC -20
#define SCOM_TBL_MALLOC -21
#define SCOM_INDEX_MALLOC -22
#define SCOM_SEN_PTR_MALLOC -23
#define SCOM_PTR_MALLOC -24
#define POT_INFO_IDF_ELE_NOT_FOUND -25
#define POT_INFO_IDF_MANY_BYTES -26
#define POT_INFO_IDF_TBL_NUM -27
#define POT_INFO_IDF_CON_NUM -28
#define POT_INFO_IDF_NO_ENTRY -29
#define CCOM_MATCH_MALLOC -30
#define CCOM_VAL_MALLOC -31
#define CONV_MODE_MISMATCH -32
#define GET_ACTION_MALLOC -33
#define POS_NOT_FOUND -34
#define FILE_POS_MODE -35
#define POS_DATA_READ_ERROR -36
#define POS_HDR_READ_ERROR -37
#define POS_HDR_MALLOC -38
#define POS_HDR_REALLOC -39
#define PBACK_LOS -40
#define PBACK_NEXT_FILE -41
#define RHDR_HDR_READ_ERROR -42
#define RHDR_HDR_MALLOC -43
#define RHDR_HDR_REALLOC -44
#define FILL_HEADER -45
#define DREC_NOT_FOUND -46
#define CONV_CAL_MALLOC -47
#define DREC_NO_FILES -48
#define DREC_READ_ERROR -49
#define DREC_HDR_READ_ERROR -50
#define DREC_HDR_MALLOC -51
#define DREC_HDR_REALLOC -52
#define RESET_EULER_REALLOC -53
#define RESET_MODE_REALLOC -54
#define RESET_NOT_FOUND -55
#define RESET_DATA_MALLOC -56
#define RESET_DATA_REALLOC -57
#define RESET_ANGLE_REALLOC -58
#define FILE_POS_EULER -59
#define TOO_MANY_EULER -60
#define ALLOC_EV_REALLOC -61
#define CRIT_TBL_NOT_FOUND -62
#define CONST_ANG_MALLOC -63
#define CONST_TEMP_MALLOC -64
#define CNVT_NOT_FOUND -65
```
```c
#define TBL_MISC_MALLOC -66
#define TBL_MALLOC -67
#define TBL_IDF_ELE_NOT_FOUND -68
#define TBL_IDF_MANY_BYTES -69
#define TBL_IDF_TBL_NUM -70
#define TBL_IDF_CON_NUM -71
#define TBL_IDF_NO_ENTRY -72
#define UPDATE_IDF_ELE_NOT_FOUND -73
#define UPDATE_IDF_MANY_BYTES -74
#define UPDATE_IDF_TBL_NUM -75
#define UPDATE_IDF_CON_NUM -76
#define UPDATE_IDF_NO_ENTRY -77
#define SEN_IDF_ELE_NOT_FOUND -78
#define SEN_IDF_MANY_BYTES -79
#define SEN_IDF_TBL_NUM -80
#define SEN_IDF_CON_NUM -81
#define SEN_IDF_NO_ENTRY -82
#define ONCE_IDF_ELE_NOT_FOUND -83
#define ONCE_IDF_MANY_BYTES -84
#define ONCE_IDF_TBL_NUM -85
#define ONCE_IDF_CON_NUM -86
#define ONCE_IDF_NO_ENTRY -87
#define ONCE_DATA_MALLOC -88
#define ONCE_D_TYPE_MALLOC -89
#define ONCE_TBL_INFO_MALLOC -90
#define ONCE_CTARGET_MALLOC -91
#define ONCE_CLEN_MALLOC -92
#define ONCE_TDW_LEN_MALLOC -93
#define ONCE_SEN_STAT_MALLOC -94
#define BAD_SCPOT_FORMAT -95
#define BAD_PA_FORMAT -96
#define MODE_TBL_SZ_IDF_ELE_NOT_FOUND -97
#define MODE_TBL_SZ_IDF_MANY_BYTES -98
#define MODE_TBL_SZ_IDF_TBL_NUM -99
#define MODE_TBL_SZ_IDF_CON_NUM -100
#define MODE_TBL_SZ_IDF_NO_ENTRY -101
#define EXP_IDF_ELE_NOT_FOUND -102
#define EXP_IDF_MANY_BYTES -103
#define EXP_IDF_TBL_NUM -104
#define EXP_IDF_CON_NUM -105
#define EXP_IDF_NO_ENTRY -106
#define CRIT_IDF_ELE_NOT_FOUND -107
#define CRIT_IDF_MANY_BYTES -108
#define CRIT_IDF_TBL_NUM -109
#define CRIT_IDF_CON_NUM -110
#define CRIT_IDF_NO_ENTRY -111
```
#define NEW_IDF_ELE_NOT_FOUND -112
#define NEW_IDF_MANY_BYTES -113
#define NEW_IDF_TBL_NUM -114
#define NEW_IDF_CON_NUM -115
#define NEW_IDF_NO_ENTRY -116
#define NEW_SCALE_MALLOC -117
#define CONV_CAL_VECTOR_MISMATCH -118
#define TIMING_MALLOC -119
#define PBACK_NO_HEADER -120
#define PBACK_NO_DATA -121
#define CHK_DATA_NOT_FOUND -122
#define NUM_CAL_REALLOC -123
#define FILE_POS_DATA_GAP -124
#define READ_IN_MALLOC -125
#define READ_IN_IDF_ELE_NOT_FOUND -126
#define READ_IN_IDF_MANY_BYTES -127
#define READ_IN_IDF_TBL_NUM -128
#define READ_IN_IDF_CON_NUM -129
#define READ_IN_IDF_NO_ENTRY -130
#define CREATE_TBL_MALLOC -131
#define CREATE_IDF_ELE_NOT_FOUND -132
#define CREATE_IDF_MANY_BYTES -133
#define CREATE_IDF_TBL_NUM -134
#define CREATE_IDF_CON_NUM -135
#define CREATE_IDF_NO_ENTRY -136
#define CONST_IDF_ELE_NOT_FOUND -137
#define CONST_IDF_MANY_BYTES -138
#define CONST_IDF_TBL_NUM -139
#define CONST_IDF_CON_NUM -140
#define CONST_IDF_NO_ENTRY -141
#define SET_VWIDTH_BAND_MALLOC -142
#define SET_VWIDTH_CENTER_MALLOC -143
#define BAD_VFMT -144
#define DKEY_PROJECT -145
#define DKEY_MISSION -146
#define DKEY_EXPERIMENT -147
#define DKEY_INSTRUMENT -148
#define DKEY_VINST -149
#define FILL_SEN_NOT_FOUND -150
#define FILL_SEN_MALLOC -151
#define FILL_SEN_REALLOC -152
#define FILL_SEN_BASE_MALLOC -153
#define FILL_SEN_BASE_REALLOC -154
#define FILL_SEN_TBL_MALLOC -155
#define FILL_BIN_MISSING -156
#define FILL_NOT_FOUND -157
#define FILL_ARRAY_MALLOC -158
#define FILL_UNITS_MALLOC -159
#define FILL_UNITS_REALLOC -160
#define FILL_SWP_MALLOC -161
#define FILL_SWP_REALLOC -162
#define FILL_DATA_MALLOC -163
#define FILL_INFO_MALLOC -164
#define NO_EMPTY_BUFFERS -165
#define UNITS_IND_NOT_FOUND -166
#define UNITS_NO_SENSOR -167
#define UNITS_NO_MATCH -168
#define COLLAPSE_NOT_FOUND -169
#define COLLAPSE_MALLOC -170
#define COLLAPSE_SEN_MALLOC -171
#define COLLAPSE_DATA_MALLOC -172
#define THETA_CHK_MALLOC -173
#define THETA_BIN_MALLOC -174
#define ORDER_THETA_MALLOC -175
#define PHI_DIFF_UNITS -176
#define FILL_PHI_FIRST -177
#define FILL_PHI_LAST -178
#define THETA_DIFF_UNITS -179
#define CDIMEN_NOT_FOUND -180
#define CDIMEN_COLLAPSE -181
#define CDIMEN_MANY_SCAN -182
#define CENTER_NOT_FOUND -183
#define CENTER_NO_SENSOR -184
#define BPTR_NOT_FOUND -185
#define CENTER_TMP_MALLOC -186
#define BAND_MALLOC -187
#define CENTER_MALLOC -188
#define SET_BIN_NOT_FOUND -189
#define SET_BIN_MALLOC -190
#define SET_BIN_INDEX_MALLOC -191
#define SET_BIN_BAD_FMT -192
#define CALC_TRES_NOT_FOUND -193
#define CALC_CENTER_DREC -194
#define RET_PHI_NOT_FOUND -195
#define CPTR_RET_PHI -196
#define NO_RET_PHI -197
#define FILL_THETA_NOT_FOUND -198
#define FILL_THETA_COLLAPSE -199
#define IMAGE_NOT_FOUND -200
#define IMAGE_READ_ERROR -201
#define IMAGE_HDR_MALLOC -202
#define IMAGE_HDR_REALLOC -203
#define NEXT_FILE_TIME_NOT_FOUND -204
#define NEXT_FILE_TIME_FILE_OPEN -205
#define NEXT_FILE_TIME_INFO_DUP -206
#define PA_MAIN_DATA_MISSING -207
#define SET_SCAN_NOT_FOUND -208
#define SCAN_BIN_MISSING -209
#define SET_SCAN_TBL_MALLOC -210
#define SCAN_INDEX_MALLOC -211
#define SCAN_IDF_ELE_NOT_FOUND -212
#define SCAN_IDF_MANY_BYTES -213
#define SCAN_IDF_TBL_NUM -214
#define SCAN_IDF_CON_NUM -215
#define SCAN_IDF_NO_ENTRY -216
#define BUF_BIN_NOT_FOUND -217
#define FILL_DISC_NOT_FOUND -218
#define FILL_DISC_BIN_MISSING -219
#define FILL_DISC_NO_PHI -220
#define FILL_DISC_MALLOC -221
#define DISC_DATA_MALLOC -222
#define DISC_TMP_MALLOC -223
#define TBL_VAR_NOT_CAL -224
#define TBL_VAR_NOT_RAW -225
#define CNVT_BAD_TBL_OPER -226
#define CNVT_NO_TMP -227
#define CNVT_TMP_MALLOC -228
#define CNVT_BAD_TBL_NUM -229
#define MODE_PTR_MALLOC -230
#define MODE_TBL_MISC_MALLOC -231
#define MODE_TBL_IDF_ELE_NOT_FOUND -232
#define MODE_TBL_IDF_MANY_BYTES -233
#define MODE_TBL_IDF_TBL_NUM -234
#define MODE_TBL_IDF_CON_NUM -235
#define MODE_TBL_IDF_NO_ENTRY -236
#define MODE_TBL_VAR_NOT_CAL -237
#define MODE_TBL_VAR_NOT_RAW -238
#define MODE_TBL_MALLOC -239
#define ASCII_AFTER_SENSOR -240
#define CONV_MODE_BAD_MODE -241
#define CONV_MODE_BAD_TBL_NUM -242
#define FILL_SEN_MODE_TYPE -243
#define UNITS_IND_MODE_TYPE -244
#define MODE_INFO_NOT_FOUND -245
#define MODE_INFO_MALLOC -246
#define MODE_INFO_REALLOC -247
#define MODE_INFO_BASE_MALLOC -248
#define MODE_INFO_BASE_REALLOC -249
#define MODE_INFO_TBL_MALLOC -250
#define MODE_INFO_NO_MODES -251
#define MODE_UNITS_IND_NOT_FOUND -252
#define MODE_UNITS_FILE_OPEN -253
#define MODE_UNITS_INFO_DUP -254
#define MODE_UNITS_NO_MODE -255
#define MODE_UNITS_NO_MATCH -256
#define FILL_MODE_ARRAY_MALLOC -257
#define MODE_UNITS_MALLOC -258
#define MODE_UNITS_REALLOC -259
#define MODE_DATA_MALLOC -260
#define FILL_MODE_NOT_FOUND -261
#define FILL_MODE_FILE_OPEN -262
#define FILL_MODE_INFO_DUP -263
#define MODES_NOT_REQUESTED -264
#define ALLOC_MODE_INFO_MALLOC -265
#define MODE_FILE_OPEN -266
#define MODE_INFO_DUP -267
#define CRIT_ACT_MALLOC -268
#define FILL_BASE_TIME_MISSING -269
#define FILL_DISC_BASE_TIME_MISSING -270
#define FILL_MODE_BASE_TIME_MISSING -271
#define CREATE_DATA_MALLOC -272
#define CREATE_DATA_ALL_MALLOC -273
#define CREATE_DATA_ALL_REALLOC -274
#define RESET_PITCH_MALLOC -275
#define RESET_PITCH_REALLOC -276
#define PITCH_MALLOC -277
#define PA_TBL_MALLOC -278
#define SWEEP_TIME_MALLOC -279
#define PINFO_IDF_ELE_NOT_FOUND -280
#define PINFO_IDF_MANY_BYTES -281
#define PINFO_IDF_TBL_NUM -282
#define PINFO_IDF_CON_NUM -283
#define PINFO_IDF_NO_ENTRY -284
#define FILE_POS_PA -285
#define PA_BAD_SRC -286
#define PA_BAD_FRAC -287
#define PA_UNIT_NORMAL -288
#define PA_UNIT_MALLOC -289
#define PA_DATA_MALLOC -290
#define CNVT_BAD_DTYPE -291
#define PA_BAD_TIMES -292
#define SWEEP_NOT_FOUND -293
#define SWEEP_BIN_MISSING -294
#define SWEEP_UNITS_MALLOC -295
#define SWEEP_UNITS_REALLOC -296
#define SWEEP_SWP_MALLOC -297
#define SWEEP_SWP_REALLOC -298
#define SWEEP_DATA_MALLOC -299
#define SWEEP_INFO_MALLOC -300
#define SWEEP_WITH_FILL -301
#define FILL_WITH_SWP -302
#define SMODE_UNITS_MALLOC -303
#define SMODE_UNITS_REALLOC -304
#define SMODE_DATA_MALLOC -305
#define ALLOC_SMODE_INFO_MALLOC -306
#define SWEEP_MODE_NOT_FOUND -307
#define SWEEP_MODE_FILE_OPEN -308
#define SWEEP_MODE_INFO_DUP -309
#define SWEEP_MODES_NOT_REQUESTED -310
#define SWEEP_MODE_WITH_FILL -311
#define FILL_WITH_SWP_MODE -312
#define SWEEP_DISC_NOT_FOUND -313
#define SWEEP_DISC_BIN_MISSING -314
#define SWEEP_DISC_NO_PHI -315
#define SWEEP_DISC_WITH_FILL -316
#define FILL_WITH_SWP_DISC -317
#define RESET_DCOS_MALLOC -318
#define RESET_DCOS_VAL_MALLOC -319
#define RESET_DCOS_VAL_REALLOC -320
#define MASS_PA_ERROR -321
#define CHRG_PA_ERROR -322
#define RET_FPTR_NOT_FOUND -323
#define RET_CBPTR_NOT_FOUND -324
#define RET_CBPTR_NO_SENSOR -325
#define RET_CENTER_NOT_FOUND -326
#define NO_PA_CONSTANT -327
#define NPHI_NOT_FOUND -328
#define NPHI_NO_BINS -329
#define NBINS_NOT_FOUND -330
#define NBINS_NO_BINS -331
#define WRONG_DATA_STRUCTURE -332
#define VIDF_OPEN_PTR_MALLOC -333
#define VIDF_OPEN_PTR_REALLOC -334
#define FILL_ENV_NOT_FOUND -335
#define FILL_ENV_BASE_TIME_MISSING -336
#define FILL_ENV_BIN_MISSING -337
#define FILL_ENV_WITH_SWP -338
#define CP_MAIN_DATA_MISSING -339
#define CP_BAD_SRC -340
#define CP_BAD_SRC -341
#define BAD_CP_FORMAT -342
#define COMPUTE_MOMENTS -343
#define CP_INFO_IDF_ELE_NOT_FOUND -344
#define CP_INFO_IDF_MANY_BYTES -345
#define CP_INFO_IDF_TBL_NUM -346
#define CP_INFO_IDF_CON_NUM -347
#define CP_INFO_IDF_NO_ENTRY -348
#define ONCE_CSCOPE_MALLOC -349
#define SET_VWIDTH_UPPER_BAND_MALLOC -350
#define ONCE_BAD_HEADER_FMT -351
#define ONCE_BAD_TENSOR_RANK -352
#define ONCE_BAD_TENSOR_LENGTHS -353
#define HDR_FMT_ONE_MALLOC -354
#define HDR_FMT_TWO_MALLOC -355
#define HDR_FMT_TWO_DQUAL -356
#define UPDATE_IDF_BAD_PA_DEF -357
#define UPDATE_IDF_NO_FILL -358
#define CREATE_TENSOR_DATA_ALL_MALLOC -359
#define CREATE_TENSOR_DATA_ALL_REALLOC -360
#define HDR_FMT_ONE_MALLOC -361
#define CREATE_TENSOR_DATA_MALLOO -362
#define WRONG_HEADER_FORMAT -363
#define TENSOR_NOT_FOUND -364
#define TENSOR_NO_FILES -365
#define TENSOR_READ_ERROR -366
#define TENSOR_DQUAL_MALLOC -367
#define TENSOR_DQUAL_REALLOC -368
#define TENSOR_HDR_READ_ERROR -369
#define TENSOR_HDR_MALLOC -370
#define TENSOR_HDR_REALLOC -371
#define TENSOR_MODE_MALLOC -372
#define TENSOR_MODE_REALLOC -373
#define TENSOR_DATA_MALLOC -374
#define TENSOR_DATA_REALLOC -375
#define ONCE_BAD_NUM_TBLS -376
#define ONCE_BAD_CAL_TARGET -377
#define TENSOR_DATA_TDW_LEN -378
#define CHK_TDATA_NOT_FOUND -379
#define COLLAPSE_DATA_ADDRESS -380
#define NEG_BIN_STAT -381
#define FIRST_SEN_NOT_FOUND -382
#define ONCE_BAD_SWP_LEN -383
#define ONCE_BAD_DA_METHOD -384
#define ONCE_BAD_SEN_MODE -385
#define TENSOR_DATA_TDW_LEN -386
#define CHK_TDATA_NOT_FOUND -387
#define START_SPIN_NOT_FOUND -388
#define START_SPIN_NO_POS -388
#define START_SPIN_NO_SPIN -389
#define START_SPIN_ETIME -390
#define START_SPIN_MALLOC -391
#define START_SPIN_ALL_MALLOC -392
#define ONCE_SPIN_OFF_MALLOC -393
#define TURN_OFF_PA_NOT_FOUND -394
#define TURN_ON_EA_NOT_FOUND -395
#define START_ELE_BAD_SENSOR -396
#define START_ELE_SPIN_NO_SENSOR -397
#define READ_SPIN_NOT_FOUND -398
#define READ_SPIN_NO_START -399
#define READ_SPIN_ALL_REALLOC -400
#define READ_SPIN_PARTIAL -401
#define SPIN_INFO_MALLOC -402
#define SPIN_UNITS_MALLOC -403
#define SPIN_UNITS_REALLOC -404
#define SPIN_SWP_MALLOC -405
#define SPIN_SWP_REALLOC -406
#define SPIN_DATA_MALLOC -407
#define SPIN_DATA_PIX_NOT_FOUND -408
#define SPIN_DATA_PIX_WITH_FILL_SWEEP -409
#define SPIN_DATA_PIX_BIN_MISSING -410
#define SPIN_DATA_PIX_CENTER_BAND_MISSING -411
#define SPIN_DATA_PIX_NO_SPIN -412
#define SPIN_DATA_NOT_FOUND -413
#define SPIN_DATA_BIN_MISSING -414
#define SPIN_DATA_WITH_FILL_SWEEP -415
#define SPIN_DATA_CENTER_BAND_MISSING -416
#define SPIN_DATA_NO_SPIN -417
#define FILL_CENTER_BAND_MISSING -418
#define FILL_ENV_CENTER_BAND_MISSING -419
#define FILL_DISC_CENTER_BAND_MISSING -420
#define SWEEP_CENTER_BAND_MISSING -421
#define SWEEP_DISC_CENTER_BAND_MISSING -422
#define NEW_BAD_TBL_OFFSET -423
#define UPDATE_IDF_BAD_SPIN_DEF -424
#define SPIN_SRC_MALLOC -425
#define START_SPIN_TIME_MALLOC -426
#define SPIN_SRC_BAD_SRC -427
#define SPIN_SINFO_IDF_ELE_NOT_FOUND -428
#define SPIN_SINFO_IDF_MANY_BYTES -429
#define SPIN_SINFO_IDF_TBL_NUM -430
#define SPIN_SINFO_IDF_CON_NUM -431
#define SPIN_SINFO_IDF_NO_ENTRY -432
#define FILE_POS_SPIN -433
```c
#define SPIN_SRC_MAIN_DATA_MISSING -434
#define READ_SPIN_DSRC_READ -435
#define READ_SPIN_DSRC_BACK_SPIN -436
#define READ_SPIN_SENSOR_NOT_FOUND -437
#define ONCE_CDTYPE_MALLOC -438
#define CP_STR_MALLOC -439
#define CP_DATA_MALLOC -440
#define FILE_POS_CP -441
#define CNVT_NO_ADV -442
#define CNVT_BAD_BUF_NUM -443
#define UPDATE_IDF_BAD_BUF_NUM -444
#define POT_TBL_MALLOC -446
#define POT_BAD_SRC -447
#define POT_BAD_FRAC -448
#define POT_MALLOC -449
#define TURN_ON_CP_NOT_FOUND -450
#define POT_DATA_MALLOC -451
#define FILE_POS_POT -452
#define POT_BAD_TIMES -453
#define RESET_POT_REALLOC -455
#define UPDATE_IDF_BAD_PMI_DEF -456
#define SWP_TIMES_TMP_MALLOC -457
#define READ_IN_BAD_TBL_OFFSET -458
#define CREATE_BAD_TBL_OFFSET -459
#define OVERRIDE_NOT_FOUND -460
#define OVERRIDE_NO_POT -461
#define UPDATE_IDF_BAD_PM1_DEF -462
#define OVERRIDE_TOO_MANY_POT_TBLS -463
#define OVERRIDE_TBL_FMT_MALLOC -464
#define OVERRIDE_BAD_TBL_FMT_VALUE -465
#define RESET_CP_REALLOC -466
#define OVERRIDE_NO_POT_TBLS -467
#define OVERRIDE_NO_POT_TBLS -468
#define OVERRIDE_TOO_MANY_POT_TBLS -469
#define OVERRIDE_TBL_FMT_MALLOC -470
#define OVERRIDE_BAD_TBL_FMT_VALUE -471
#define OVERRIDE_TBL_FMT_MALLOC -472
#define OVERRIDE_BAD_TBL_FMT_VALUE -473
#define OVERRIDE_BAD_TBL_FMT_VALUE -474
#define OVERRIDE_BAD_TBL_FMT_VALUE -475
#define OVERRIDE_BAD_TBL_FMT_VALUE -476
#define OVERRIDE_BAD_TBL_FMT_VALUE -477
#define OVERRIDE_BAD_TBL_FMT_VALUE -478
#define OVERRIDE_BAD_TBL_FMT_VALUE -479
```
```c
#define MORE_EULER_CONSTANT_AXIS -480
#define EULER_IDF_DATA_MALLOC -481
#define EULER_BAD_SRC -482
#define EULER_BAD_TIMES -483
#define EULER_BAD_FRAC -484
#define EULER_INFO_IDF_ELE_NOT_FOUND -485
#define EULER_INFO_IDF_MANY_BYTES -486
#define EULER_INFO_IDF_TBL_NUM -487
#define EULER_INFO_IDF_CON_NUM -488
#define EULER_INFO_IDF_NO_ENTRY -489
#define TRANS_3D_BINNED_MALLOC -490
#define DESTROY_NO_IDF_DATA -491
#define DESTROY_NO_TENSOR_DATA -492
#define FILE_POS_INVALID_DATA -493
#define BKGD_MAIN_DATA_MISSING -494
#define BKGD_BAD_SRC -495
#define BKGD_TBL_MALLOC -496
#define BKGD_MALLOC -497
#define BKGD_DATA_MALLOC -498
#define BKGD_IDF_DATA_MALLOC -499
#define BKGD_INFO_IDF_ELE_NOT_FOUND -500
#define BKGD_INFO_IDF_MANY_BYTES -501
#define BKGD_INFO_IDF_TBL_NUM -502
#define BKGD_INFO_IDF_CON_NUM -503
#define BKGD_INFO_IDF_NO_ENTRY -504
#define FILE_POS_BKG -505
#define BKGD_BAD_TIMES -506
#define RESET_BKG_REALLOC -507
#define BKGD BAD_FRAC -508
#define UPDATE_IDF_BAD_BKG_DEF -509
#define NO_BKG CONSTANT -510
#define UTIL_START_CODE -1
#define UTIL_STOP_CODE -511
```
**USER_DEFS.H**

function - defines mnemonics that can be utilized for coding purposes

**SYNOPSIS**

#include "user_defs.h"

**DESCRIPTION**

The *user_defs.h* include file holds mnemonics available for use with some of the IDFS routines. All mnemonics are initialized through a `define` statement. The user may include this file and use the mnemonics to help improve the readability of the calling sequence for some of the IDFS routines.

**CONTENTS OF FILE**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR</td>
<td>1</td>
</tr>
<tr>
<td>SWEEP_STEP</td>
<td>2</td>
</tr>
<tr>
<td>CAL_DATA</td>
<td>3</td>
</tr>
<tr>
<td>MODE</td>
<td>4</td>
</tr>
<tr>
<td>D_QUAL</td>
<td>5</td>
</tr>
<tr>
<td>PITCH_ANGLE</td>
<td>6</td>
</tr>
<tr>
<td>START_AZ_ANGLE</td>
<td>7</td>
</tr>
<tr>
<td>STOP_AZ_ANGLE</td>
<td>8</td>
</tr>
<tr>
<td>SC_POTENTIAL</td>
<td>9</td>
</tr>
<tr>
<td>BACKGROUND</td>
<td>10</td>
</tr>
<tr>
<td>MAX_DATA_TYPE</td>
<td>10</td>
</tr>
<tr>
<td>MAX_UNITS_BUFFERS</td>
<td>9</td>
</tr>
<tr>
<td>OUTSIDE_MIN</td>
<td>-3.4e38</td>
</tr>
<tr>
<td>OUTSIDE_MAX</td>
<td>3.4e38</td>
</tr>
<tr>
<td>VALID_MIN</td>
<td>-3.0e38</td>
</tr>
<tr>
<td>VALID_MAX</td>
<td>3.0e38</td>
</tr>
<tr>
<td>UNTouched_BUFFER</td>
<td>-2</td>
</tr>
<tr>
<td>FREE_BUFFER</td>
<td>-1</td>
</tr>
<tr>
<td>PARTIAL_WORKING</td>
<td>0</td>
</tr>
<tr>
<td>BUFFER_READY</td>
<td>1</td>
</tr>
<tr>
<td>NUM_BUFFERS</td>
<td>5</td>
</tr>
<tr>
<td>LEADING_EDGE</td>
<td>1</td>
</tr>
<tr>
<td>TRAILING_EDGE</td>
<td>0</td>
</tr>
<tr>
<td>MASS_DIMEN</td>
<td>1</td>
</tr>
<tr>
<td>PHI_DIMEN</td>
<td>2</td>
</tr>
<tr>
<td>THETA_DIMEN</td>
<td>3</td>
</tr>
<tr>
<td>SCAN_DIMEN</td>
<td>4</td>
</tr>
<tr>
<td>DATA_DIMEN</td>
<td>5</td>
</tr>
<tr>
<td>NO_DIMEN</td>
<td>6</td>
</tr>
<tr>
<td>CHARGE_DIMEN</td>
<td>7</td>
</tr>
<tr>
<td>SCAN_INDEX</td>
<td>0</td>
</tr>
<tr>
<td>THETA_INDEX</td>
<td>1</td>
</tr>
<tr>
<td>PHI_INDEX</td>
<td>2</td>
</tr>
</tbody>
</table>
#define MASS_INDEX 3
#define CHARGE_INDEX 4
#define SCALAR_INDEX 5
#define DIMEN_OFF 0
#define DIMEN_ON 1
#define DIMEN_CONSTANT 2
#define NO_AVG 1
#define STRAIGHT_AVG 2
#define STRAIGHT_INT 3
#define SPHERICAL_INT 4
#define STRAIGHT_AVG_AZ 5
#define FLUX_INT 6
#define MOMENTS_INT 7
#define POINT_INT 1
#define BAND_INT 2
#define FIXED_Sweep 1
#define VARIABLE_Sweep 2
#define ZERO_SPACING 0
#define LIN_SPACING 1
#define LOG_SPACING 2
#define VARIABLE_SPACING 3
#define POINT_STORAGE 1
#define BAND_STORAGE 2
#define NO_BIN_FILL 1
#define LIN_ROW_COL 2
#define LIN_COL_ROW 3
#define CON_ROW_COL 4
#define CON_COL_ROW 5
#define LEAST_SQ_Fit 6
#define TORAD 0.017453292519943
#define PA_NOT_APPLICABLE 0
#define PA_READY 1
#define PA_DB_ERROR 2
#define PA_DATA_MISSING 3
#define PA_IR_ERROR 4
#define SPIN_SRC_NOT_APPLICABLE 0
#define SPIN_SRC_READY 1
#define SPIN_SRC_DB_ERROR 2
#define SPIN_SRC_DATA_MISSING 3
#define SPIN_SRC_IR_ERROR 4
#define POT_NOT_APPLICABLE 0
#define POT_READY 1
#define POT_DB_ERROR 2
#define POT_DATA_MISSING 3
#define POT_IR_ERROR 4
#define EULER_NOT_APPLICABLE 0
#define EULER_READY 1
#define EULER_DB_ERROR 2
#define EULER_DATA_MISSING 3
#define EULER_IR_ERROR 4
#define CP_NOT_APPLICABLE 0
#define CP_READY 1
#define CP_DB_ERROR 2
#define CP_DATA_MISSING 3
#define CP_IR_ERROR 4
#define BKGD_NOT_APPLICABLE 0
#define BKGD_READY 1
#define BKGD_DB_ERROR 2
#define BKGD_DATA_MISSING 3
#define BKGD_IR_ERROR 4
#define NO_SPECIFIED_CS -1
#define SPACECRAFT_CS 0
#define PMI_CS 1
#define GEI_CS 2
#define GEO_CS 3
#define GSE_CS 4
#define GSM_CS 5
#define SM_CS 6
#define MAG_CS 7
#define HEE_CS 8
#define HAE_CS 9
#define HEEQ_CS 10
LIBTREC_IDFS.H
function - contains prototypes for the IDFS routines that return time-averaged, sample-
averaged, or spin-averaged data

SYNOPSIS
#include "libtrec_idfs.h"

DESCRIPTION
The libtrec_idfs.h include file contains the ANSI C prototypes for the IDFS routines that
are used to retrieve data that is time-averaged, sample-averaged or spin-averaged. These
routines can be found in the 2R section of the IDFS Programmers Manual. This file should
be included in the source code wherever an IDFS routine is called from to ensure that the
correct number of arguments are used and to ensure that the argument types match.

SEE ALSO
libbase_idfs  1H
SCF_codes (3H)

function - defines possible return values and associated mnemonics for the SCF software

SYNOPSIS
#include "SCF_codes.h"

DESCRIPTION
The SCF_codes.h include file holds all of the defined return values for the SCF routines defined in sections 3R and 4R. All return values are associated with a mnemonic through a define statement. The user of the SCF routines should include this file and use the mnemonics defined for the return values. This allows for the values to be changed in the future with no code modification.

CONTENTS OF FILE
#define ALL_OKAY 1
#define SCF_TERMINATE 45
#define NO_SCF_FILE -3001
#define SCF_CONTACT_MALLOC -3002
#define SCF_COMMENTS_MALLOC -3003
#define SCF_INPUT_MALLOC -3004
#define SCF_INPUT_TBL_MALLOC -3005
#define SCF_TEMP_MALLOC -3006
#define SCF_OUTPUT_VAR_MALLOC -3007
#define SCF_MAP_MALLOC -3008
#define SCF_EQNS_MALLOC -3009
#define SCF_EQNS_REALLOC -3010
#define LOCATE_SCF_NOT_FOUND -3011
#define LOCATE_SCF_MALLOC -3012
#define LOCATE_SCF_REALLOC -3013
#define READ_SCF_NO_TOKEN -3014
#define READ_SCF_BAD_DSRC -3015
#define READ_SCF_BAD_INPUT -3016
#define READ_SCF_BAD_TEMP -3017
#define READ_SCF_BAD_OUTPUT -3018
#define READ_SCF_BAD_FIELD -3019
#define READ_SCF_NO_DIMEN -3020
#define READ_SCF_BAD_EQNS -3021
#define READ_SCF_BAD_TOKEN -3022
#define SCF_NO_FUNCTION -3023
#define READ_SCF_BAD_FUNCTION -3024
#define READ_SCF_BAD_INDEX -3025
#define READ_SCF_ELSE_INFO -3026
#define SCF_ACQ_MANY_READS -3027
#define SCF_OUTPUT_DATA_STR -3028
#define SCF_FRAC_MALLOC -3029
#define SCF_ARGS_MALLOC -3030
#define SCF_MATRIX_MALLOC -3031
#define SCF_OPEN_ERROR -3032
#define SCF_POS_ERROR -3033
#define SCF_SAMP_POS -3034
#define SCF_SAMP_BAD_RATE -3035
#define SCF_FAST_BAD_LOCATE -3036
#define SCF_ALLOC_PLOT_LOC -3037
#define SCF_REALLOC_PLOT_LOC -3038
#define SCF_PROCESS_BAD_EX -3039
#define SCF_BAD_FRAC -3040
#define SCF_INDEX_MALLOC -3042
#define SCF_ARG_RANK -3043
#define SCF_RES_RANK -3044
#define SCF_NUM_ARGS -3045
#define SCF_CREATE_ALL_MALLOC -3046
#define SCF_CREATE_ALL_REALLOC -3047
#define SCF_CREATE_MALLOC -3048
#define SCF_OUTPUT_MALLOC -3049
#define SCF_NO_FUNC_IN_LIB -3050
#define SCF_OUTPUT_CALC -3051
#define SCF_OPEN_RTIME -3052
#define SCF_INDEX_MALLOC -3053
#define SCF_INVALID_INDEX -3054
#define SCF_BAD_LOGICAL_OPER -3055
#define SCF_SAMP_BAD_INPUT_NUM -3056
#define SCF_SAMP_BAD_ACCUM -3057
#define SCF_SAMP_BAD_LOCATE -3058
#define SCF_SAMP_SWP_MALLOC -3059
#define SCF_SAMP_VECTOR_ACCUM -3060
#define SCF_SAMP_VECTOR_SRC -3061
#define SCF_NON_VOID -3062
#define SCF_VOID -3063
#define SCF_NO_INDEX -3064
#define SCF_BREAK_STMT -3065
#define SCF_DIMEN_MALLOC -3066
#define SCF_SIZE_MISMATCH -3067
#define SCF_AORDER_MISMATCH -3068
#define SCF_RORDER_MISMATCH -3069
#define SCF_TSIZE_MISMATCH -3070
#define SCF_TENSOR_MANY_ARGS -3071
#define SCF_TENSOR_MANY_ARGS -3072
#define SCF_TENSOR_MANY_ARGS -3073
#define SCF_TENSOR_MANY_ARGS -3074
#define SCF_TDIMEN -3075
#define SCF_TSUM_VDIMEN -3076
#define SCF_TSUM_Row_DIMEN -3077
#define SCF_TSUM_RSIZE -3078
#define SCF_TWDIMEN -3079
#define SCF_TWVDIMEN -3080
#define SCF_TWSUM_ROW_DIMEN -3081
#define SCF_TWSUM_COL_DIMEN -3082
#define SCF_TWSUM_RSIZE -3083
#define SCF_TW_WLEN -3084
#define SCF_TENSOR_SAME_RANK -3085
#define SCF_MASK_LENGTHS -3086
#define SCF_TEXTRACT_SDIMEN -3087
#define SCF_TEXTRACT_START -3088
#define SCF_TEXTRACT_RES_RANK -3089
#define SCF_TEXTRACT_RES_DIMEN -3090
#define SCF_TEXTRACT_INDEX -3091
#define SCF_TINSERT_SDIMEN -3092
#define SCF_TINSERT_START -3093
#define SCF_TINSERT_INDEX -3094
#define SCF_TINSERT_SIZE -3095
#define SCF_TINT_CLEN -3096
#define SCF_TSPACE -3097
#define SCF_TENSOR_VECTOR_SRC -3098
#define SCF_FILL_SZ -3099
#define SCF_NO_LIBRARY -3100
#define SCF_BIN_BAD_SWP_FMT -3101
#define SCF_BIN_BAD_FMT -3102
#define SCF_BIN_BAD_OVAR_NUM -3103
#define SCF_BIN_MALLOC -3104
#define SCF_BIN_BAD_CNUM -3105
#define SCF_BIN_BAD_BNUM -3106
#define SCF_BIN_CLENGTH -3107
#define SCF_BIN_BLENGTH -3108
#define SCF_BIN_BAD_VFMT -3109
#define SCF_CENTER_MALLOC -3110
#define SCF_BAND_MALLOC -3111
#define SCF_SELECT_OVAR_NUM -3112
#define SCF_SELECT_BIN_MISSING -3113
#define SCF_SELECT_MALLOC -3114
#define SCF_SELECT_DEF_MALLOC -3115
#define SCF_SELECT_DEF_REALLOC -3116
#define SCF_SELECT_DVAR_NUM -3117
#define SCF_SELECT_DVAR_LENGTH -3118
#define SCF_SELECT_BAND_MALLOC -3119
#define SCF_TAVG_NO_BASE_TIME -3120
#define SCF_TAVG_BIN_MISSING -3121
#define SCF_TAVG_SELECT_MISSING -3122
#define SCF_TIME_WITH_SAMPLE -3123
```c
#define SCF_AVG_STR_MALLOC -3124
#define SCF_TINFO_MALLOC -3125
#define SCF_TDATA_MALLOC -3126
#define SCF_SINFO_MALLOC -3127
#define SCF_SDATA_MALLOC -3128
#define SCF_ALG_START_NO_SAMPLE -3129
#define SCF_OINDEX_OVAR_NUM -3130
#define SCF_OINDEX_NO_AVG -3131
#define SCF_OINDEX_NO_OUTPUT -3132
#define SCF_OINDEX_NO_MATCH -3133
#define SCF_OCENTER_OVAR_NUM -3134
#define SCF_OCENTER_NO_AVG -3135
#define SCF_OCENTER_SELECT_MISSING -3136
#define SCF_NO_EMPTY_BUFFERS -3137
#define SCF_SAVG_BIN_MISSING -3138
#define SCF_SAVG_SELECT_MISSING -3139
#define SCF_SAMPLE_WITH_TIME -3140
#define SCF_NO_CENTER_VALUES -3141
#define SCF_BAD_START_STOP -3142
```
SCF_DEF.S.H

function - defines mnemonics that can be utilized for SCF coding purposes

SYNOPSIS

#include "SCF_defs.h"

DESCRIPTION

The SCF_defs.h include file holds mnemonics available for use with some of the SCF routines. All mnemonic are initialized through a define statement. The user may include this file and use the mnemonics to help improve the readability of the calling sequence for some of the SCF routines. This allows for transparent future modifications in their values.

CONTENT OF FILE

#define SCF_MEASURE_TM 1
#define SCF_MEASURE_LAT_TM 2
#define SCF_DELTA_T 1
#define USE_INPUT_VAR 2
#define USE_DELTA_T 3
#define S_BEFORE_START 1
#define S_EQUAL_START 2
#define S_AFTER_START 3
#define S_NUM_SPECIAL 15
#define SPIN_RATE 10
#define DATA_ACCUM_MS 11
#define DATA_ACCUM_NS 12
#define DATA_LAT_MS 13
#define DATA_LAT_NS 14
#define SCF_LINEAR_BINS 1
#define SCF_LOG_BINS 2
#define SCF_POINT_STORAGE 1
#define SCF_BAND-STORAGE 2
#define SCF_LESS_THAN_MIN -1
#define SCF_GREATER_THAN_MAX -2
SCF_FILE_DEFS.H

function - defines all possible mnemonics used to access elements in the SCF file

SYNOPSIS

#include "SCF_file_defs.h"

DESCRIPTION

The SCF_file_defs.h include file holds all of the defined mnemonics that can be utilized to access the data elements found within the SCF file. All mnemonics are assigned a value through a define statement. It is recommended that the read_scf routine be invoked using the defined mnemonics. This allows for transparent future modifications in their values.

CONTENTS OF FILE

#define NOT_USED 0
#define S_TITLE 0
#define S_NUM_CONTACT 1
#define S_CONTACT 2
#define S_NUM_COMMENTS 3
#define S_COMMENTS 4
#define S_NUM_INPUT 5
#define S_INPUT_NAME 6
#define S_INPUT_PROJ 7
#define S_INPUT_MISSION 8
#define S_INPUT_EXP 9
#define S_INPUT_INST 10
#define S_INPUT_VINST 11
#define S_INPUT_KEY 12
#define S_INPUT_DTYPE 13
#define S_INPUT_DNUM 14
#define S_INPUT_CSET 15
#define S_INPUT_NUM_TBLS 17
#define S_INPUT_TBLS 18
#define S_INPUT_OPERS 19
#define S_INPUT_LOWER_CUT 20
#define S_INPUT_UPPER_CUT 21
#define S_INPUT_QUAL_MIN 22
#define S_INPUT_QUAL_MAX 23
#define S_NUM_TEMP 24
#define S_TEMP_NAME 25
#define S_TEMP_DIMENSION 26
#define S_TEMP_LENGTHS 27
#define S_NUM_OUTPUT 28
#define S_OUTPUT_NAME 29
#define S_OUTPUT_DIMENSION 30
#define S_OUTPUT_LENGTHS 31
#define S_NUM_EQNS 32
```c
#define S_EQUATIONS 33
#define S_EQN_TYPE 34
#define S_EQN_START 35
#define S_EQN_STOP 36
#define S_ELSE_START 37
#define S_ELSE_STOP 38
#define SCF_EQN 0
#define SCF_FN 1
#define SCF_FOR 2
#define SCF_IF 3
#define SCF_IF_ELSE 4
#define SCF_BREAK 5
```
LIBBASE_SCF.H
function- contains prototypes for the basic set of SCF routines

SYNOPSIS
#include "libbase_SCF.h"

DESCRIPTION
The libbase_SCF.h include file contains the ANSI C prototypes for the basic set of SCF routines that interpret the contents of the SCF file, execute the algorithm defined in the SCF file and return the results of the algorithm. These routines can be found in section 3R of the IDFS Programmers Manual. This include file should be included in the source code wherever an SCF routine is called from to ensure that the correct number of arguments are used and to ensure that the argument types match.

SEE ALSO
libavg_SCF 4H
LIBAVG_SCF.H
function - contains prototypes for the SCF routines that return time-averaged or sample-averaged data

SYNOPSIS
#include "libavg_SCF.h"

DESCRIPTION
The libavg_HSCF include file contains the ANSI C prototypes for the SCF routines that are used to retrieve data for output variables that have been time-averaged or sample-averaged. These routines can be found in the 4R section of the IDFS Programmers Manual. This include file should be included in the source code wherever an SCF routine is called from to ensure that the correct number of arguments are used and to ensure that the argument types match.

SEE ALSO
libbase_SCF 3H
IDF_DATA
  function - IDFS data structure

SYNOPSIS
  #include "util_str.h"

  struct idf_data
  {
    SDDAS_ULONG  data_key;
    SDDAS_SHORT  header_format;
    SDDAS_SHORT  sensor;
    SDDAS_SHORT  byear;
    SDDAS_SHORT  bday;
    SDDAS_LONG   bmilli;
    SDDAS_LONG   bnano;
    SDDAS_LONG   bsec;
    SDDAS_LONG   bnsec;
    SDDAS_SHORT  eyear;
    SDDAS_SHORT  eday;
    SDDAS_LONG   emilli;
    SDDAS_LONG   enano;
    SDDAS_LONG   esec;
    SDDAS_LONG   ensec;
    SDDAS_SHORT  mode_byear;
    SDDAS_SHORT  mode_bday;
    SDDAS_LONG   mode_bmilli;
    SDDAS_LONG   mode_bnano;
    SDDAS_SHORT  mode_eyear;
    SDDAS_SHORT  mode_eday;
    SDDAS_LONG   mode_emilli;
    SDDAS_LONG   mode_enano;
    SDDAS_LONG   data_accum_ms;
    SDDAS_LONG   data_accum_ns;
    SDDAS_LONG   data_lat_ms;
    SDDAS_LONG   data_lat_ns;
    SDDAS_LONG   swp_reset_ms;
    SDDAS_LONG   swp_reset_ns;
    SDDAS_LONG   sen_reset_ms;
    SDDAS_LONG   sen_reset_ns;
    SDDAS_FLOAT  *start_az;
    SDDAS_FLOAT  *stop_az;
    SDDAS_FLOAT  start_theta;
    SDDAS_FLOAT  stop_theta;
    SDDAS_FLOAT  *pitch_angles;
    SDDAS_FLOAT  *potential;
    SDDAS_FLOAT  *background;
  
SDDAS_SHORT  num_swp_steps;
SDDAS_USHORT  num_sample;
SDDAS ULONG cal_len;
SDDAS_USHORT num_angle;
SDDAS_USHORT num_pitch;
SDDAS_USHORT num_potential;
SDDAS_USHORT num_background;
SDDAS_LONG sun_sen;
SDDAS_LONG spin_rate;
SDDAS_LONG *cal_data;
SDDAS_LONG *sen_data;
SDDAS_LONG *swp_data;
SDDAS_LONG *mode;
SDDAS_LONG d_qual;
SDDAS_UINT cal_size;
SDDAS_UINT data_size;
SDDAS_UINT swp_size;
SDDAS_UINT mode_size;
SDDAS_UINT angle_size;
SDDAS_UINT pitch_size;
SDDAS_UINT potential_size;
SDDAS_UINT background_size;
SDDAS UCHAR mode_len;
SDDAS CHAR hdr_change;
SDDAS_CHAR exten[3];
SDDAS_CHAR filled_data;
SDDAS USHORT version;
SDDAS ULONG *cset_num;
struct direction_cos *dir_cosines;
struct transformation_info *idfs_transformation;
void *base_cal;
void *base_data;
void *base_swp;
void *base_angle;
void *base_mode;
void *base_pitch;
void *base_cset;
void *base_dir_cosines;
void *base_potential;
void *base_background;
void *base_transform;
};
ELEMENT DEFINITIONS

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_key</td>
<td>unique value which indicates the data set of interest</td>
</tr>
<tr>
<td>header_format</td>
<td>used to identify conventional IDFS data from multi-dimensional tensor IDFS data</td>
</tr>
<tr>
<td>sensor</td>
<td>the sensor identification number for the current data</td>
</tr>
<tr>
<td>byear</td>
<td>the year at the start of the accumulation of the first data value in the data sweep</td>
</tr>
<tr>
<td>bday</td>
<td>the day at the start of the accumulation of the first data value in the data sweep</td>
</tr>
<tr>
<td>bmilli</td>
<td>the milliseconds of the day at the start of the accumulation of the first data value in the data sweep</td>
</tr>
<tr>
<td>bnano</td>
<td>the remaining nanoseconds of the day at the start of the accumulation of the first data value in the data sweep</td>
</tr>
<tr>
<td>bsec</td>
<td>the start time of the first data value in the data sweep in seconds (bmilli + bnano)</td>
</tr>
<tr>
<td>bnsec</td>
<td>the remaining nanoseconds of the start time of the first data value in the data sweep (bmilli + bnano)</td>
</tr>
<tr>
<td>eyear</td>
<td>the year at the end of the accumulation period of the last data value in the data sweep</td>
</tr>
<tr>
<td>eday</td>
<td>the day at the end of the accumulation period of the last data value in the data sweep</td>
</tr>
<tr>
<td>emilli</td>
<td>the milliseconds of the day at the end of the accumulation period (including any latency) of the last data value in the data sweep</td>
</tr>
<tr>
<td>enano</td>
<td>the remaining nanoseconds of the day at the end of the accumulation period (including any latency) of the last data value in the data sweep</td>
</tr>
<tr>
<td>esec</td>
<td>the end time of the last data value in the data sweep in seconds (emilli + enano)</td>
</tr>
<tr>
<td>ensec</td>
<td>the remaining nanoseconds of the end time of the last data value in the data sweep (emilli + enano)</td>
</tr>
<tr>
<td>mode_byear</td>
<td>the year at the start of the accumulation for the instrument status values</td>
</tr>
<tr>
<td>mode_bday</td>
<td>the day at the start of the accumulation for the instrument status values</td>
</tr>
<tr>
<td>mode_bmilli</td>
<td>the milliseconds of the day at the start of the accumulation for the instrument status values</td>
</tr>
<tr>
<td>mode_bnano</td>
<td>the remaining nanoseconds of the day at the start of the accumulation for the instrument status values</td>
</tr>
<tr>
<td>mode_eyear</td>
<td>the year at the end of the accumulation period for the instrument status values</td>
</tr>
<tr>
<td>mode_eday</td>
<td>the day at the end of the accumulation period for the instrument status values</td>
</tr>
<tr>
<td>mode_emilli</td>
<td>the milliseconds of the day at the end of the accumulation period (including any latency) for the instrument status values</td>
</tr>
</tbody>
</table>
mode_enano - the remaining nanoseconds of the day at the end of the accumulation period (including any latency) for the instrument status values

data_accum_ms - the amount of time for a single data acquisition, in milliseconds

data_accum_ns - the remainder of data_accum_ms, in nanoseconds

data_lat_ms - the amount of dead time between successive data acquisitions, in milliseconds

data_lat_ns - the remainder of data_lat_ms, in nanoseconds

swp_reset_ms - the amount of dead time at the end of an instrument sweep, in milliseconds

swp_reset_ns - the remainder of swp_reset_ms, in nanoseconds

sen_reset_ms - the amount of dead time between successive sensor sets, in milliseconds

sen_reset_ns - the remainder of sen_reset_ms, in nanoseconds

start_az - pointer to the first element in the initial azimuthal sample angle array, with one value per data sample returned

stop_az - pointer to the first element in the final azimuthal sample angle array, with one value per data sample returned

start_theta - the initial theta angle for the sensor in question

stop_theta - the final theta angle for the sensor in question

pitch_angles - pointer to the first element in the pitch angle array, with one value per data sample returned

potential - pointer to the first element in the spacecraft potential array, with one value per data sample returned

background - pointer to the first element in the background data array, with one value per data sample returned

num_swp_steps - the number of elements returned in the sweep array

*swp_data

num_sample - the number of elements returned in the sensor data array

*sen_data

cal_len - the number of elements returned in the calibration array

*cal_data

num_angle - the number of elements returned in the azimuthal angle arrays

*start_az and *stop_az

num_pitch - the number of elements returned in the pitch angle array

*pitch_angles

num_potential - the number of elements returned in the spacecraft potential array

*potential

num_background - the number of elements returned in the background data array

*background

sun_sen - the time of the last 0º crossing

spin_rate - the current spin rate of the virtual instrument

cal_data - pointer to the first element in the calibration array

sen_data - pointer to the first element in the sensor data array

swp_data - pointer to the first element in the sweep array
mode - pointer to the first element in the mode flags array
d_qual - value which indicates the quality of the data being returned and also serves as an offset into the qual_names array defined in the VIDF file. The user should refer to the IDFS File System Definition Document for an explanation of qual_names and d_qual. The values returned are dependent upon the data set being processed.
cal_size - the number of bytes allocated for the calibration array
data_size - the number of bytes allocated for the sensor data array
swp_size - the number of bytes allocated for the sweep array
mode_size - the number of bytes allocated for the mode flags array
angle_size - the number of bytes allocated for both of the azimuthal angle arrays
pitch_size - the number of bytes allocated for the pitch angle array
potential_size - the number of bytes allocated for the spacecraft potential array
background_size - the number of bytes allocated for the background array
mode_len - the number of elements returned in the mode flags array
*mode
hdr_change - flag which indicates a header change occurred
  0 - a header change was not encountered during the retrieval of data
  1 - a header change was encountered during the retrieval of data
exten - two character extension identifying the IDFS data file set being utilized (should be a null string "" when using the default IDFS data sets)
filled_data - flag which indicates if data was placed into the data arrays for the sensor in question
  0 - data was not placed into the arrays for the sensor in question
  1 - data was placed into the arrays for the sensor in question
version - IDFS data set identification number which allows for multiple openings of the same data set
cset_num - pointer to the first element in the calibration set size array
dir_cosines - pointer to a structure which holds the direction cosines array and ancillary theta start / stop angles and azimuthal start / stop angles associated with the pitch angle data source
idfs_transformation - pointer to a structure which holds data pertinent to coordinate system transformations (currently, this includes euler angle data and/or celestial position angle data)
base_cal - the base address of the allocated memory for the calibration array
base_data - the base address of the allocated memory for the sensor data array
base_swp - the base address of the allocated memory for the sweep array
base_angle - the base address of the allocated memory for the azimuthal angle arrays
base_mode - the base address of the allocated memory for the mode flags
base_pitch - the base address of the allocated memory for the pitch angle array
base_cset - the base address of the allocated memory for the calibration set size array
base_dir_cosines - the base address of the allocated memory for the direction cosine data structure
base_potential - the base address of the allocated memory for the spacecraft potential data
base_background - the base address of the allocated memory for the background data
base_transform - the base address of the allocated memory for the coordinate system transformation structure

DESCRIPTION
The idf_data structure holds all of the currently returned data values, in raw format, from the last call to the read_drec routine.

SEE ALSO
read_drec 1R
read_drec_spin 1R
create_data_structure 1R
create_idf_data_structure 1R
direction_cos 1H
transformation_info 1H
DIRECTION_COS

type - data structure which holds the direction cosine information returned
within the idf_data IDFS data structure

SYNOPSIS

#include "util_str.h"

struct direction_cos {
    SDDAS_FLOAT *dir_cos123;
    SDDAS_FLOAT *start_az123;
    SDDAS_FLOAT *stop_az123;
    SDDAS_FLOAT *start_theta123;
    SDDAS_FLOAT *stop_theta123;
    void *base_mem;
};

ELEMENT DEFINITIONS

dir_cos123 - pointer to the first element in the direction cosines array, with
the three components being returned for each data sample
returned in the idf_data structure

start_az123 - pointer to the first element in the initial azimuthal sample
angle array, with the three components being returned for
each data sample returned in the idf_data structure

stop_az123 - pointer to the first element in the final azimuthal sample angle
array, with the three components being returned for each data
sample returned in the idf_data structure

start_theta123 - pointer to the first element in the initial theta angle array, with
the three components being returned for each data sample
returned in the idf_data structure

stop_theta123 - pointer to the first element in the final theta angle array, with
the three components being returned for each data sample
returned in the idf_data structure

base_mem - pointer to the memory allocated to hold all of the data
elements contained within the structure

DESCRIPTION

The direction_cos data structure is an integral part of the idf_data structure; that is, it is not
returned as a stand-alone data structure by the IDFS data access routines. This structure
contains the direction cosine values for each of the three components, along with ancillary
angular information pertinent to the IDFS data source utilized as the pitch angle data source.
The data is stored as triplets (x, y, z), with the 3 values being laid down sequentially within
the arrays ([0,1,2] [3,4,5] …). There is one triplet for each sample returned within the
idf_data structure. For example, if num_sample were set to 8, there would be 8 triplets
returned in the five arrays described above. Each array would contain 24 (8 * 3) values.
SEE ALSO

read_drec            1R
read_drec_spin       1R
idf_data             1H
TRANSFORMATION_INFO
function - data structure which holds the coordinate system transformation information returned within the idf_data IDFS data structure

SYNOPSIS
#include "util_str.h"

struct transformation_info
{
    SDDAS_FLOAT  *euler_angles;
    SDDAS_SHORT  *euler_rot_axis;
    SDDAS_USHORT  num_euler;
    SDDAS_UINT  euler_size;
    void    *base_euler;
    SDDAS_FLOAT  *declination_angles;
    SDDAS_FLOAT  *rt_ascension_angles;
    SDDAS_USHORT  num_celestial;
    SDDAS_UINT  celestial_size;
    void    *base_celestial;
};

ELEMENT DEFINITIONS

euler_angles  - pointer to the first element in the euler angle array, with num_euler value(s) being returned for each data sample returned in the idf_data structure

euler_rot_axis  - pointer to the first element in the euler rotation axis array, with num_euler value(s) returned

num_euler  - the number of euler angles defined

euler_size  - the number of bytes allocated for the euler angle and euler rotation axis arrays

base_euler  - the base address of the allocated memory for the euler angle and euler rotation axis data

declination_angles  - pointer to the first element in the declination angle array, with one value being returned for each data sample returned in the idf_data structure

rt_ascension_angles  - pointer to the first element in the right ascension angle array, with one value being returned for each data sample returned in the idf_data structure

num_celestial  - the number of elements returned in the celestial position angle arrays *declination_angles and *rt_ascension_angles

celestial_size  - the number of bytes allocated for the declination angle and right ascension angle arrays

base_celestial  - the base address of the allocated memory for the declination angle and right ascension angle data
DESCRIPTION
The transformation_info data structure is an integral part of the idf_data structure; that is, it is not returned as a stand-alone data structure by the IDFS data access routines. This structure contains the euler angle data and the celestial position angle data (declination and right ascension angles) pertinent to the IDFS data source being returned in the idf_data structure.

For the celestial position angle data, there is an angle returned for each sample returned within the idf_data structure; therefore, num_celestial from the transformation_info structure and num_sample from the idf_data structure are identical. However, the data is stored differently for the euler angle information. For the euler_angles data, there are num_euler values being laid down sequentially within the array for each sample returned within the idf_data structure. For example, if num_sample were set to 8 and num_euler were set to 2, there would be 16 (8 * 2) values returned, with the first 2 values associated with the first data sample, the next 2 values associated with the second data sample, etc. This is not the case for the euler_rot_axis array. Since the euler rotation axis does not change from data sample to data sample, there are num_euler elements returned in the euler_rot_axis array.

SEE ALSO
read_drec 1R
read_drec_spin 1R
idf_data 1H

EXAMPLES
The following code fragment demonstrates how to access the data within the transformation_info structure, if pertinent to the data source being processed. The code fragments assumes that a data structure has already been created and data has been read and is ready to be processed.

#include "libbase_idfs.h"

struct idf_data *EXP_DATA;
struct transformation_info *trans_ptr;
register SDDAS_USHORT loop, k;
SDDAS_USHORT offset;
void *idf_data_ptr;

EXP_DATA = (struct idf_data *) idf_data_ptr;
trans_ptr = EXP_DATA->idfs_transformation;
/* Any coordinate system transformation information defined? */

if (trans_ptr != NULL)
{
    /* Print out the euler data and rotation axis for each data sample. */

    offset = 0;
    for (k = 0; k < EXP_DATA->num_sample; ++k)
    {
        for (loop = 0; loop < trans_ptr->num_euler; ++loop, ++offset)
            printf("%05.2f %d ", *(trans_ptr->euler_angles + offset),
                      *(trans_ptr->euler_rot_axis + loop));

        printf("\n");
    }

    /* Print out the celestial position angle data. */

    for (loop = 0; loop < trans_ptr->num_celestial; ++loop)
        printf("%05.2f %05.2f \n", *(trans_ptr->declination_angles + loop),
                       *(trans_ptr->rt_ascension_angles + loop));

    printf("\n");
}
SCF_DATA
function - data structure that returns information for the output variables defined by the SCF

SYNOPSIS
#include "SCF.h"

struct scf_data
{
    SDDAS_SHORT byear;
    SDDAS_SHORT bday;
    SDDAS_LONG bmilli;
    SDDAS_LONG bnano;
    SDDAS_SHORT eyear;
    SDDAS_SHORT eday;
    SDDAS_LONG emilli;
    SDDAS_LONG enano;
    SDDAS_LONG num_output;
    SDDAS_LONG *output_length;
    SDDAS_LONG *output_index;
    void   *output_data;
    SDDAS_CHAR filename[SCF_FILENAME];
};

ELEMENT DEFINITIONS
byear    - the year at the start of the time period processed
bday    - the day of year at the start of the time period processed
bmilli    - the milliseconds of the day at the start of the time period processed
bnano    - the remaining nanoseconds of the day at the start of the time period processed
eyear    - the year at the end of the time period processed
eday    - the day of year at the end of the time period processed
emilli    - the milliseconds of the day at the end of the time period processed
enano    - the remaining nanoseconds of the day at the end of the time period processed
num_output    - the number of output variables being returned
output_length    - pointer to the first element in an array which specifies the number of data values returned for each output variable
output_index    - pointer to the first element in an array of index values (offsets) that are used to access the data in the output_data array
output_data    - pointer to the first element in the output data array
DESCRIPTION

The **scf_data** structure holds the results from the last execution of the algorithm defined for the SCF being processed. The user must call the routine **create_scf_data_structure** once for each distinct SCF file being processed. It is these structures which are expected in any SCF routine where output variable values are being retrieved.

SEE ALSO

- scf_output_data 3R
- create_scf_data_structure 3R
TENSOR_DATA
  function – multi-dimensional IDFS data structure

SYNOPSIS
  #include "util_str.h"

  struct tensor_data
  {
    SDDAS_ULONG     data_key;
    SDDAS_SHORT    header_format;
    SDDAS_SHORT    sensor;
    SDDAS_SHORT    byear;
    SDDAS_SHORT    bday;
    SDDAS_LONG     bmilli;
    SDDAS_LONG     bnano;
    SDDAS_LONG     bsec;
    SDDAS_LONG     bnsec;
    SDDAS_SHORT    eyear;
    SDDAS_SHORT    eday;
    SDDAS_LONG     emilli;
    SDDAS_LONG     enano;
    SDDAS_LONG     esec;
    SDDAS_LONG     ensec;
    SDDAS_SHORT    mode_byear;
    SDDAS_SHORT    mode_bday;
    SDDAS_LONG     mode_bmilli;
    SDDAS_LONG     mode_bnano;
    SDDAS_SHORT    mode_eyear;
    SDDAS_SHORT    mode_eday;
    SDDAS_LONG     mode_emilli;
    SDDAS_LONG     mode_enano;
    SDDAS_LONG     data_accum_ms;
    SDDAS_LONG     data_accum_ns;
    SDDAS_LONG     data_lat_ms;
    SDDAS_LONG     data_lat_ns;
    SDDAS_LONG     swp_reset_ms;
    SDDAS_LONG     swp_reset_ns;
    SDDAS_LONG     sen_reset_ms;
    SDDAS_LONG     sen_reset_ns;
    SDDAS_SHORT    tensor_rank;
    SDDAS_LONG     tensor_sizes[IDFS_MAX_DIMEN];
    SDDAS_ULONG    tnext_dimen[IDFS_MAX_DIMEN];
    SDDAS_ULONG    num_vals;
    SDDAS_ULONG    cal_len;
    SDDAS_LONG     sun_sen;
    SDDAS_LONG     spin_rate;
  }
SDDAS_LONG  *sen_data;
SDDAS_LONG  *mode;
SDDAS_LONG  *d_qual;
SDDAS_LONG  *cal_data;
SDDAS_UINT  data_size;
SDDAS_UINT  tensor_bytes;
SDDAS_UINT  mode_size;
SDDAS_UINT  dqual_size;
SDDAS_UINT  cal_size;
SDDAS_ULONG  num_dqual;
SDDAS_UCHAR  mode_len;
SDDAS_CHAR  hdr_change;
SDDAS_CHAR  exten[3];
SDDAS_CHAR  filled_data;
SDDAS_USHORT  version;
SDDAS_FLOAT  *tdata;
SDDAS_FLOAT  *tcaldata;
void    *base_data;
void    *base_tdata;
void    *base_cal;
void    *base_tcaldata;
void    *base_mode;
void    *base_dqual;

};

ELEMENT DEFINITIONS

data_key   - unique value which indicates the multi-dimensional data set
of interest
header_format - used to identify conventional IDFS data from multi-
dimensional tensor IDFS data
sensor   - the sensor identification number for the current data
byear   - the year at the start of the accumulation of the first data value
bday   - the day at the start of the accumulation of the first data value
bmilli   - the milliseconds of the day at the start of the accumulation of
the first data value
bnano   - the remaining nanoseconds of the day at the start of the
accumulation of the first data value
bsec   - the start time of the first data value in seconds
(bmilli + bnano)
bnsec   - the remaining nanoseconds of the start time of the first data
value (bmilli + bnano)
eyear   - the year at the end of the accumulation period of the last data
value
eday   - the day at the end of the accumulation period of the last data
value
emilli - the milliseconds of the day at the end of the accumulation period (including any latency) of the last data value
esec - the end time of the last data value in seconds (emilli + enano)
ensec - the remaining nanoseconds of the end time of the last data value (emilli + enano)
enano - the remaining nanoseconds of the day at the end of the accumulation period (including any latency) of the last data value
mode_byear - the year at the start of the accumulation for the instrument status values
mode_bday - the day at the start of the accumulation for the instrument status values
mode_bmilli - the milliseconds of the day at the start of the accumulation for the instrument status values
mode_bnano - the remaining nanoseconds of the day at the start of the accumulation for the instrument status values
mode_eyear - the year at the end of the accumulation period for the instrument status values
mode_eday - the day at the end of the accumulation period for the instrument status values
mode_emilli - the milliseconds of the day at the end of the accumulation period (including any latency) for the instrument status values
mode_enano - the remaining nanoseconds of the day at the end of the accumulation period (including any latency) for the instrument status values
data_accum_ms - the amount of time for a single data acquisition, in milliseconds
data_accum_ns - the remainder of data_accum_ms, in nanoseconds
data_lat_ms - the amount of dead time between successive data acquisitions, in milliseconds
data_lat_ns - the remainder of data_lat_ms, in nanoseconds
swp_reset_ms - the amount of dead time at the end of an instrument sweep, in milliseconds
swp_reset_ns - the remainder of swp_reset_ms, in nanoseconds
sen_reset_ms - the amount of dead time between successive sensor sets, in milliseconds
sen_reset_ns - the remainder of sen_reset_ms, in nanoseconds
tensor_rank - the number of dimensions returned in the data tensor
tensor_sizes - an array of size tensor_rank that holds the maximum lengths of each of the dimensions defined
tnext_dimen - an array of size tensor_rank that holds the number of data values to bypass to get to the next index for a given dimension ([0] = 1st dimension or slowest varying dimension)
num_vals - the number of elements returned in the sensor data tensor
cal_len - the number of elements returned in the calibration data tensor
sun_sen - the time of the last 0º crossing
spin_rate - the current spin rate of the virtual instrument
sen_data - pointer to the first element in the sensor data tensor
mode - pointer to the first element in the mode flags array
d_qual - pointer to the first element in the data quality tensor whose value indicates the quality of the data being returned and also serves as an offset into the qual_names array defined in the VIDF file. The user should refer to the IDFS File System Definition Document for an explanation of qual_names and d_qual.

cal_data - pointer to the first element in the calibration data tensor
data_size - the number of bytes allocated for the sensor data tensor
tensor_bytes - the number of bytes allocated to convert the tensor data according to d_type specified in the VIDF
mode_size - the number of bytes allocated for the mode flags array
dqual_size - the number of bytes allocated for data quality tensor
cal_size - the number of bytes allocated for the calibration data tensor
num_dqual - the number of elements returned in the data quality vector
mode_len - the number of elements returned in the mode flags array

*mode

hdr_change - flag which indicates a header change occurred
0 - a header change was not encountered during the retrieval of data
1 - a header change was encountered during the retrieval of data

exen - two character extension identifying the multi-dimensional IDFS data file set being utilized (should be a null string """) when using the default IDFS data sets

filled_data - flag which indicates if data was placed into the data tensor for the sensor in question
0 - data was not placed into the tensor for the sensor in question
1 - data was placed into the tensor for the sensor in question

version - IDFS data set identification number which allows for multiple openings of the same data set
tdata - pointer to the first element in the sensor data tensor which has been converted according to d_type in the VIDF
tcaldata - pointer to the first element in the calibration data tensor which has been converted according to d_type in the VIDF
base_data - the base address of the allocated memory for the sensor data tensor
base_tdata - the base address of the allocated memory for the sensor data that is converted according to d_type in the VIDF
base_cal - the base address of the allocated memory for the calibration data tensor
base_tcaldata - the base address of the allocated memory for the calibration data that is converted according to d_type in the VIDF
base_mode - the base address of the allocated memory for the mode flags
tensor
base_dqual - the base address of the allocated memory for the data quality tensor

DESCRIPTION

The tensor_data structure holds all of the currently returned data values, in raw format, from the last call to the read_drec_tensor routine. The only conversion that is performed on the data is a possible conversion from integer storage to its true floating point value, as defined by d_type in the VIDF file.

SEE ALSO

read_drec_tensor 1R
create_data_structure 1R
create_tensor_data_structure 1R