LIS/OTD Software Guide

Dennis J. Boccippio
Kevin Driscoll
John Hall
Dennis Buechler
Global Hydrology and Climate Center
977 Explorer Blvd
Huntsville, AL 35806

June 2, 1998

Contents

Part I

Introduction

Chapter 1

Introduction

This document serves as a guide to the software intended for use with satellite data from the Optical Transient Detector (OTD) and Lightning Imaging Sensor (LIS). The software suite consists of both fully featured GUI (Graphical User Interface) driven applications, and collections of high- and low-level APIs (Application Programming Interfaces). The software is designed to simplify, as much as possible, user access to the OTD and LIS lightning data sets, which are currently distributed in HDF (Hierarchical Data Format) files. The suite is designed with four goals in mind: *simplicity*, *reusability*, *compatibility* and *deployment*. By providing software strongly tailored to these goals, we hope to minimize each user's time spent accessing and managing the datasets, and maximize the time spent actually *analyzing* them.

1.1 Background

Lightning data from sensors deployed by the NASA / Marshall Space Flight Center began in April 1995, with the prototype OTD instrument, and continues with the launch of the LIS instrument aboard the Tropical Rainfall Measurement Mission (TRMM). The collection, processing, filtering, quality assurance and archiving of these data are nontrivial processes, and the datasets themselves require a fair amount of ancillary information in order to be useful (instrument viewtime, alert flags, navigation data, etc.). Rather than maintain these various data in separate data streams (e.g., one file for lightning locations, one for viewing information, one for navigation data), we have chosen to store all data for each satellite orbit in HDF format files.

The HDF structure is a flexible, self-describing data format, which allows collections of different data (both in 'type', e.g., integer, floating point, etc, and 'order', e.g., point, vector, array) to be stored together in the same data file. HDF is an extremely useful tool in scientific data management, but until

quite recently has been hobbled by a rather cumbersome, low-level programming interface required to access the data in each file. Given the relative ease with which scientists can create confusing arrangements of data within the HDF files, the result can easily be a rather unattractive vehicle for storing and deploying data.

The ultimate goal of the LIS/OTD software suite is to allow scientists to "forget" that the data is even in HDF format. We have observed that in recent years, far more time is often spent writing "translator" and "extraction" code for various datasets than is actually spent analyzing them. Furthermore, for large, complicated, internally linked data sets, students are often scared off by the complexities of memory allocation, pointer management, etc., which might be required in some programming languages to handle these data sets.

We thus provide a variety of tools for scientists to access our data at the level at which they feel comfortable, or may require. The simplest data interface is a "conventional" menu-driven software application, which requires little to no programming skill. This is provided in the form of an IDL program. The next simplest interface is a "high-level" programming API, or set of libraries. "High level" APIs are those which, for our purposes, allow easy import of the contents of an HDF file into a programming language such as C or IDL, effectively hiding all of the "low level" HDF input and output. This frees the scientist from writing I/O routines, and allows him or her to concentrate on writing analysis algorithms. Finally, "Low level" APIs are provided which allow more detailed access to the HDF data, for programmers who feel comfortable with such access. Even these APIs are still "higher level" than the lowest level interface functions provided in the NCSA HDF distribution, which users of LIS and OTD data should *never* need to use.

1.2 Software Strategy

As noted above, the LIS/OTD software suite includes applications, high level programming libraries, and low level programming libraries. Together, these give users the flexibility to access the data as they are able (based on their computing resources) and as they require (based on their needs, e.g., file-at-a-time analysis, batch processing, etc.). The suite has been designed with the following four goals in mind: *simplicity*, *resuability*, *compatibility*, and *deployment*.

1. **Simplicity** means that the LIS/OTD software should have a low learning curve - users should familiarize themselves with the features of the sensors themselves, and the general way in which the data is organized, but shouldn't have to develop vastly greater programming skill sets than they already possess in order to use the data. Users need only learn the software tools which they feel comfortable using. "Simplicity" also means that the LIS/OTD data should be contained in a clear, logical hierarchy

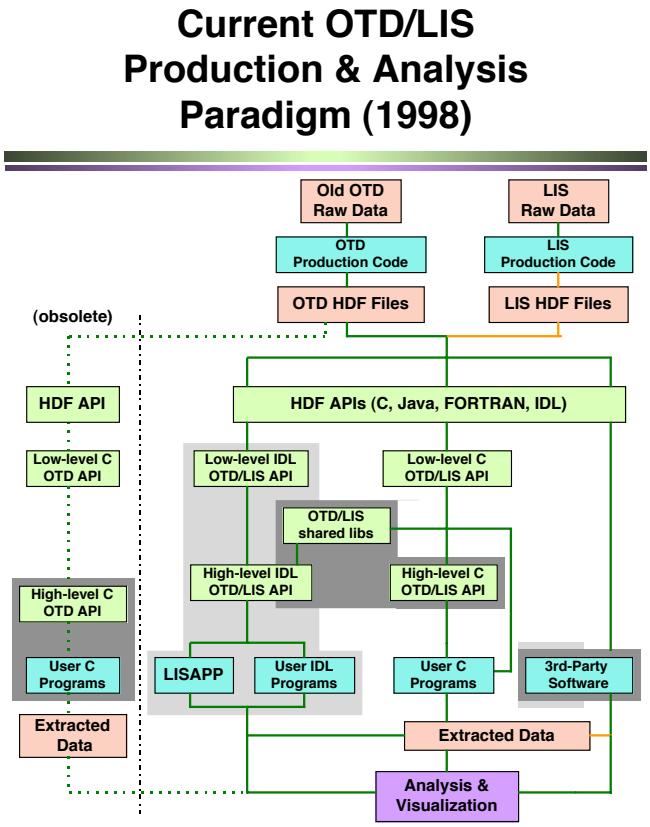
within HDF files. Significant effort has gone into improving the old OTD HDF data format to improve its clarity and ease of use.

2. **Reusability** means that the data structures, programming techniques, and much of the actual analysis code written by users can be nearly language-independent, at least in terms of data and algorithm structure. This would mean that an IDL routine written by one scientist to analyze LIS data could be easily translated to C by a scientist who did not own a copy of IDL. This is accomplished primarily by keeping the overall data structure definitions, variable names, etc. nearly identical between the HDF file and the IDL and C variables into which the software reads HDF data.

3. **Compatibility** means that, as much as possible, we strive to make newly deployed software backwards-compatible with previously deployed data formats and/or programming tools. Thus, old format OTD HDF files are completely supported by the new programming tools. Further, the simplified HDF data structure should now be general enough to be easily reused to contain level 1 data from the geostationary Lightning Mapping Sensor (LMS), when that instrument comes online.

4. **Deployment** means that our tools must be usable across the widest variety of computing and analysis platforms possible. Our goal is not to ensure that every tool will work on every platform; resources at the LIS SCF (Scientific Computing Facility) simply cannot allow that level of software support. However, we have attempted to guarantee that at least one analysis tool *is* available on every platform, regardless of the operating system, amount of commercial software installed, etc. This should be a distinct improvement over our older OTD software tools, which were severely limited in their cross-platform deployment. We want as many scientists as possible to use our data!

Fig. 1 illustrates the current implementation of the LIS/OTD software. The diagram looks complicated, but it's actually not. At the top are the raw LIS/OTD data, directly received from the satellites. Users never see these "level 0" data; they are immediately processed at the LIS/SCF with our "production code". This in-house software organizes and filters the raw satellite data, and computes some value-added parameters for good measure. The production code generates HDF data files, which are then distributed to the user community. Currently there are two HDF file formats, the old OTD format and the new LIS/OTD format. Both are readable by the software in this package. Eventually, all old OTD data will be reprocessed and stored in the new file format, but this is a detail most users need not worry about.



The ultimate goal is at the bottom of the diagram - analysis and visualization. Everything in between comprises different "analysis paths", or ways to achieve that goal. Users who have access to IDL 5.0.2 (a commercial scientific data language available for almost all computer platforms) or higher can follow the leftmost paths. Note that since IDL allows direct plotting, there is no "intermediate" ASCII data dump stage; users may immediately read in an HDF data file and begin working with it, either with the LISAPP application in this package, or with their own IDL programs using our high-level IDL API. Several other applications (such as *EOSView* or Fortner's commercial *Noesys* software) allow limited access to our HDF files, we term these "3rd party software", on the right side of the diagram. Users who do not have access to commercial software packages can still access our data through the high- and low-level C language APIs, also provided as part of this package. Analysis using C usually involves some data dumping to intermediate ASCII files, and later visualization with other software tools.

The grey shading in this diagram denotes two things: light grey indicates commercial software, and dark grey indicates tools in this package for which we

can only guarantee limited cross-platform support (although they may work on many platforms). As can be seen from the diagram: we retain backwards compatibility with OTD files, offer a variety of analysis paths, offer cross-platform deployment, offer at least one direct data-to- visualization path, offer high level APIs, and offer at least one analysis path without platform or software limitations (the low-level C APIs, which use only 'vanilla' HDF calls). Again, the selection of analysis path will be ultimately determined by the user's hardware and software resources, and their comfort level with the various programming tools.

Users interested in the future evolution of this software strategy should consult the diagrams in Appendix C.

1.3 The Tools

As discussed above, the LIS/OTD software suite includes a number of different tools, ranging from pre-built applications to software libraries and APIs. The tools are briefly summarized below.

1. **C low-level API**
   The low-level C language API includes both basic structure definitions for LIS and OTD data, and rudimentary input/output routines to extract portions of the data from LIS/OTD HDF data files, storing them in C structures. Users are responsible for properly opening and closing the HDF files, allocating memory, etc. The code is fairly generic C and HDF. Thus, this package is perhaps the most complicated, but also the most cross-platform compatible, path to access and analyze the data. The vast majority of users should, however, be able to also implement one of the higher-level APIs (C or IDL), and thus never worry about the more complicated low-level C API.

2. **C high-level API**
   The high-level C language API includes basic structure definitions for the LIS and OTD data, as well as "nested" structure definitions and simple, single-call input/output routines. These routines allow direct import of all data in a LIS/OTD HDF data file into a C language "structure of structures", whose form matches exactly the internal HDF data organization. Basic subsetting, filtering, geolocation and date/time conversion utilities are also included in this package. Users need not worry about any details of HDF file access; it is all done transparently by the I/O routines. The code is still fairly generic C, and should be usable across most Un*x variant platforms, as well as desktop PC operating systems. However, direct support by the LIS/SCF will be limited to Windows, MacOS, IRIX and

Linux operating systems.

3. **Shared libraries**
   A set of shared libraries is available for a few platforms, which allow IDL to read in LIS/OTD HDF data significantly faster than by using IDL's built-in HDF input/output routines. The performance improvement can be up to a factor of two. Support for these shared libraries is limited to MacOS, IRIX and Linux platforms, with Windows support expected by Q3-Q4 1998.

4. **IDL high-level API**
   The high-level IDL API includes basic structure definitions for LIS and OTD data, as well as "nested" structure definitions and simple, single-call input/output routines. These routines allow direct import of all data in a LIS/OTD HDF data file into an IDL language "structure of structures", whose form matches exactly the internal HDF data organization. Users may thus easily write their own custom IDL programs to analyze and visualize the LIS/OTD data. Several routines for geolocation and date/time conversion are included in the package. It is compatible with the shared libraries described above, which accelerate HDF data input. The API should work with IDL on any platform which supports IDL 5.0.2 or higher; an IDL license is of course required.

5. **LISAPP**
   LISAPP is a self-contained, GUI-based, menu-driven IDL application for analysis and visualization of LIS/OTD HDF data. It is compatible with the shared libraries described above, which accelerate HDF data input. LISAPP allows users to load partial or entire LIS/OTD HDF orbit data files, and examine the data in either tabular or graphical format. Plotting of LIS/OTD lightning data with background image overlays is supported. Export of plots to GIF format is supported, as is export of data to ASCII format. The application also leaves the inputted HDF data file resident in IDL memory, and accessible as a structure variable at the IDL command line. This allows direct, flexible, interactive analysis of individual LIS/OTD orbit data.

Tables 1.1 and 1.2 summarize the current and future computer platform support for the various programming tools included in this package. Much of the code in this package is fairly modular, and users who wish to port and test certain tools to platforms not supported below are welcome to do so.

As can be seen from the tables, direct or indirect support is available for all major computer platforms, with the exception of nearly obsolete operating systems/computers such as Macintosh 68K and Windows 3.1. No support is

currently planned for OS/2, BeOS or OpenStep. Limited C language support for LinuxPPC/Mklinux may be available by Q3-Q4 1998.

	Win 3.1	Win 95	Win NT	Linux	Mac68K	MacPPC
HDF 4.1r1		√	√	√		√
C	√	√	√	√	√	√
IDL 5.0.3		√	√	√		√
C API Lo-Lev		⋆	⋆	†		√
C API Hi-Lev		⋆	⋆	†		√
Shared Libs		⋆	⋆	⋆		√
IDL API		√	†	†		√
IDL LISAPP		√	†	†		√
Noesys EOSView		√	√			√
Commercial	*N*	*Y*	*Y*	*Y*	*N*	*Y*
Freeware	*N*	⋆	⋆	*Y*	*N*	*Y*

Table 1.1: Software availability for various workstation or mainframe computer systems. √ denotes software available and supported. ⋆ denotes software planned for release in Q3-Q4 1998, support level TBD. † denotes software which should operate on a given platform, but for which no direct support is available from the LIS/SCF.

	SunOS	Solaris	AIX	HPUX	DigUnix	IRIX	VMS
HDF 4.1r1	√	√	√	√	√	√	√
C	√	√	√	√	√	√	√
IDL	√	√	√	√	√	√	√
C API Lo-Lev	†	†	†	†	†	√	†
C API Hi-Lev	†	†	†	†	†	√	†
Shared Libs						√	
IDL API	†	†	†	†	†	√	
IDL LISAPP	†	†	†	†	†	√	
Noesys EOSView		√	√	√	√	√	
Commercial	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*
Freeware	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*

Table 1.2: Software availability for various workstation or mainframe computer systems. √ denotes software available and supported. ⋆ denotes software planned for release in Q3-Q4 1998, support level TBD. † denotes software which should operate on a given platform, but for which no direct support is available from the LIS/SCF.

Chapter 2

OTD/LIS/LMS Lightning Data

As discussed in chapter 1, the data contained in LIS/OTD HDF files includes both actual lightning locations and parameters and ancillary platform/sensor related information. In order to properly use the sensor data, a basic understanding of the instruments' deployment, characteristics and limitations is required. This chapter briefly discusses the important highlights of each sensor.

2.1 The Instruments

Three sensors comprise the NASA/MSFC spaceborne lightning detection program. All are based upon a staring charge coupled device (CCD) camera system with a narrowband interferometric filter centered on the peak lightning emission line. All three sensors search for transient optical pulses which rise above the background scene radiance. All three sensors measure individual pixel transients, then group these pulses in space and time into data units more analagous to lightning strokes and flashes. All three sensors are capable of detecting total (intracloud and cloud-to-ground) lightning during both day and night, although the lightning detection efficiency (LDE) varies from sensor to sensor.

Each sensor in the NASA/MSFC lightning program represents a specific "milestone" on the route to operational spaceborne lightning detection. The Optical Transient Detector (OTD) launched in April 1995 aboard the Microlab-1 platform was a prototype/proof-of-concept instrument, built and deployed cheaply, and intended to demonstrate the technology's viability and collect the first-ever global lightning climatology unbiased by the diurnal lightning cycles. The Lightning Imaging Sensor (LIS) launched in November 1997 aboard the Tropical Rainfall Measurement Mission (TRMM) is a full science instrument, designed to improve on OTD's hardware, and collect high-quality storm scale lightning data, from which scientific algorithms could be crafted for opera-

tional application of spaceborne lightning data. The Lightning Mapping Sensor (LMS), planned for launch aboard a future GOES satellite, will be a geostationary instrument which implements further hardware refinements, and uses the science knowledge gained from LIS for operational purposes, primarily forecasting and hazard alerts.

2.1.1 Optical Transient Detector (OTD)

OTD has been continuously monitoring global lightning from a near-polar orbit since April 1995. The instrument is comprised of a 128x128 CCD pixel array, with individual pixel resolutions from 8-13 km across, and a total field of view of 1300x1300 sq km. The instrument has about a 50% detection efficiency; see (ref) for more details. The Microlab platform precesses slowly through the diurnal cycle, and use of composite OTD data for longer time scale averages *must* take account of this, or severe aliasing of the diurnal lightning cycle will contaminate the data. 55 day averaging should be adequate to anti-alias the data. The OTD sensor attitude (orientation) may rotate over the course of an orbit, so individual geographic locations seen by OTD in a given orbit may be seen for anywhere between 1 and 270 seconds (this information is available in the HDF file). Note also that the Microlab platform's navigation (ephemeris and attitude) data is sometimes poor, resulting in low spatial accuracy. While adequate for global or regional composites, use of the data for storm-scale applications should be undertaken cautiously, with the navigation problems in mind. Further details may also be found in (ref).

2.1.2 Lightning Imaging Sensor (LIS)

The LIS instrument has been in a 35 degree (tropical) orbit since November 1997. LIS also is comprised of a 128x128 CCD pixel array, with individual pixel resolutions from 3-6 km across, and a total field of view of 550x550 sq km. The instrument has about a 90-95% detection efficiency, although exact values have yet to be determined. Note also that the instrument uses *variable thresholding* based upon the background cloud radiance (the sensitivity varies inversely with the background radiance). Since the ultimate detection efficiency is a function of the threshold applied, users should make careful use of the threshold values associated with each lightning datum. The TRMM instrument usually flies in a "x-forward" or "x-reverse" attitude (orientation), so the LIS sensor array does not rotate, and almost all geographic locations observed by LIS during a given orbit are viewed for about 90 seconds (this information is available in the HDF file). TRMM navigation is usually very good, so the LIS instrument has very high spatial accuracy. LIS data is ideal for storm-scale applications and tropical climatological usage. 100 day averaging windows should be sufficient to remove aliases of the diurnal lightning cycle from climatological LIS data.

2.2 Data Organization

In the current software suite, both OTD and LIS data are represented internally by a new data structure, which we term here the "LIS/OTD structure". This structure matches the format of the new LIS HDF files. Old OTD HDF files which are input using this software are translated, internally, to the new structure. Eventually the entire OTD dataset will be reprocessed and stored in the new format, but this change should not affect most users.

As noted previously, the LIS/OTD data structure has been crafted such that the same data hierarchy exists in the HDF data files and in the IDL and C variables used to read in this data. This hierarchy is summarized thus:

1. Orbit data
   (a) Orbit summary
   (b) Point data
      i. Point data summary
      ii. Viewtime granules
      iii. Background image summaries
      iv. Lightning data
         A. Areas
         B. Flashes
         C. Groups
         D. Events
   (c) One second data
2. Orbit metadata
   (a) Summary Image
   (b) Text metadata

In order to help understand the data organization, users should think of each orbit as having three types of information contained within it. The first is granular (*point*) data. This category includes all data which exists at a discrete geographic location and time. In our data files, this includes *viewtime granules*, *background image summaries* and *lightning* data. The *lightning* data is further composed of four different levels of optical pulse grouping, *areas*, *flashes*, *groups* and *events*. The second major type of information is continuous (*one-second*) data. This data is a continuous record over the course of the orbit of key parameters such as attitude and ephemeris, warning flags, etc. The third major type of information is orbit-descriptive *metadata*. In the LIS/OTD data representation, this includes both a raster *summary image* and ASCII *text metadata*. The organization of these three data types is further shown in Fig. 2, and each element described below.

1. **Orbit data** is actually a data "container", a placeholder to collect related information. Previously, the orbit data container was assumed to be the same as the HDF file itself, which made some programming and analysis tasks involving more than one orbit somewhat complicated. It is included in the present LIS/OTD structure for convenience and completeness.

   (a) **Orbit summary** is a small data structure containing vital information such as the orbit start and end time, and the number of one-second (see below) data structures contained within each **orbit**.

   (b) **Point data** includes any data which occurs discretely in space and time. It is also a data "container" with no values directly associated with it, merely data "children".

      i. **Point data summary** is a small data structure containing the number of occurrences of each point datum in this branch of the data tree.
      ii. **Viewtimes** are the method chosen to describe the OTD or LIS sensors' coverage of given geographic regions on the earth. The earth is divided into 0.5 degree bins for this computation. Every time the sensor begins to see a given spatial bin, a new "viewtime granule" is created. Each granule contains the bin location, the start and stop times of the coverage, and an "effective observation" time which tries to account for bins which are only partially grazed by the sensor (and thus may be shorter than the end-start time difference). Note that because of the curvature of the sensor field of view, possible rotation of the sensor, or obscuration by the OTD gravity boom, there may be multiple viewtime granules for a given lat/lon bin (i.e., the sensor 'starts' and 'stops' seeing a given location multiple times during a single overflight). Viewtimes are necessary to convert observed lightning *counts* into actual lightning *rates*.

      iii. **Background image summaries** contain the locations and times at which the LIS or OTD-observed background scenes are reported. While each sensor continuously monitors the background radiances to detect optical transients, these scenes are only recorded for archival purposes every 30 seconds or so. The background image summaries provide the basic information needed to geolocate (plot on a map) the recorded scenes, in conjunction with the one-second data. Note that the new LIS file format keeps the actual background scenes in separate HDF files, in order to keep the science dataset size manageable for users with limited disk space and no need for the background images.

iv. **Lightning** is another data container, into which are placed the various groupings of lightning optical pulses which correspond to more familiar physical features such as thunderstorms, flashes, strokes, etc.

A. **Areas** are distinct regions of the earth which have one or more flashes (see below) in a given orbit. They are meant to roughly correspond to individual thunderstorm cells, or perhaps charge centers. Note that because of resolution and sensitivity issues, areas derived from OTD data may not be strictly intercomparable with areas derived from LIS data.

B. **Flashes** are collections of observed pulse groups (see below) which are both spatially and temporally "close" to each other. They are intended to closely match the physical lightning "flash", i.e., a collection of nearby channels which may illuminate and re-illuminate multiple times. Note that with the finer resolution of LIS, separate "flashes" may actually be describing individual channel segments rather than physically distinct flashes.

C. **Groups** are collections of observed pulse events (see below) occuring during the same 2 ms time frame, which are also adjacent to each other in sensor CCD pixel space. Groups may be interpreted as individual lightning strokes or K-changes. They are the basic building blocks of flashes, and hence of areas. The algorithm which clusters events into groups is robust, well defined and stable, so users may wish to consider groups, rather than events, as the lowest level lightning data.

D. **Events** are individual sensor pixel transients, or optical pulses. These are the true basic building blocks of the reported lightning data, first collected into groups, which are then clustered into flashes, which are finally arranged in areas. Note from the sensor descriptions above that the basic pixel size varies from sensor to sensor.

(c) **One-second data** are a *continuous* record of the sensor and platform's status over the course of its orbit. They include key information such as alert flags, navigation (attitude and ephemeris) data, threshold information, etc. Certain alert summary flags contained in the point data may be investigated further by examining the relevant one-second data found here.

2. **Orbit metadata** are descriptive records summarizing various aspects of

the orbit data above. They include a raster summary image and ASCII (text) metadata.

(a) **Summary Image** is a raster image plot of the orbit in each file. It is included for quick examination and manual identification of specific orbits of interest.

(b) **Metadata** is ASCII (text) descriptive information which summarizes the contents of each file. It is primarily intended for use by data archival and subsetting systems, and not intended for end-users.

The detailed contents of each data group, as represented in HDF, C and IDL, are described in Appendix A.

2.3 Data Usage

This section contains some useful tips on working with the LIS/OTD data. We recommend users read this section carefully, to avoid false, misleading or inappropriate results when using our data.

☞ **Not all degrees are created equal.** This warning applies especially to OTD data being used climatologically. Remember that if you are working with fixed size lat/lon bins for convenience in gridding data, a "square degree" at the equator differs in size from a "square degree" near the poles. Be sure to make the appropriate corrections if you are presenting your results in square kilometers.

☞ **Use the viewtimes!** Because both LIS and OTD are in low-earth orbit, their sampling of the earth is fairly limited. Also, there are times when the sensors' hardware buffers fill up (too much lightning or noise data is being seen) and the sensor becomes briefly "blinded". Further, in the case of OTD, there are times when the instrument is either intentionally turned off or in a warmup condition, again effectively "blinding" it. All these instances are accounted for in the viewtime granule data. Whether you are examining a single storm or building a climatology, be sure to consider the actual viewtime granules recorded in the files. Satellite-observed lightning "counts" are virtually meaningless unless you convert them to lightning "rates" using the viewtimes.

☞ **Use the viewtimes sensibly!** There will be some instances where the sensor field of view just "grazes" a viewtime grid cell during a given orbit; this is a result of the finite viewtime grid resolution, the geometry of the field of view and the on-orbit rotation of sensors themselves (this rotation

is rare for LIS but very common for OTD). As a result, some total viewtimes will be very low (several seconds or less); indeed, if you use the viewtimes populated with the point data (e.g., `orbit.point.lightning.area.delta_time`; integer values) rather than the viewtime granules themselves (`orbit.point.viewtime`; floating point values), some associated viewtime times may even be zero. Clearly trying to calculate a lightning rate from a very brief observation time is ill-advised. You should definitely consider this when writing flash rate caclulation algorithms using the LIS/OTD data.

☞ **Consider variance.** This is an issue related to viewtime. Note that at the "top" and "bottom" of each sensor's orbital path, a given latitude will be seen many more times than at the equator (over time). While this can be corrected for by using the viewtime data, don't forget that this also implies that lightning flash rate estimates at the most poleward extents of the orbits have significantly *lower* variance than near the equator. Also, for OTD, data dropouts clustered in two geographic regions: the South Atlantic Anomaly (SAA) noise region, and the eastern United States (due to planned satellite resets when passing over the Orbital Sciences ground station). Climatological flash rate estimates in these regions thus will have distinctly *higher* variance than elsewhere on the earth. (Note that the SAA effects on LIS are much smaller, and hence the region of higher variance is also much smaller). These differences in variance should be considered when interpreting climatological flash rate results.

☞ **Consider thresholds.** Over its lifetime, OTD has used several different threshold settings. LIS, as noted above, constantly uses variable thresholds based upon the actual background radiance at each CCD pixel. These thresholds are recorded with each lightning datum. Since the lightning detection efficiency (LDE) can vary by up to 20% between various threshold settings, you may want to either select an LDE based upon the actual threshold used (if this is known), or postprocess the data to reject low-amplitude events and keep your dataset (be it orbital, daily, annual, etc.) at the highest of all possible thresholds used within your period of interest.

☞ **Check the alert flags.** Each lightning datum carries with it an "alert summary flag". This is a 1-byte number whose bits correspond to "warning" and "fatal" conditions for each of the following: the instrument, the platform, the environment, and the software processing. Non-zero alert flags should not be ignored! The individual bits in the alert flags, along with the TAI93 time of the event, should be used to reference the one-second data. Within the one-second data are more comprehensive summary flags which will help you track down the specific problem, and decide whether to use the datum or not. Of course, the decision of whether or not to use data flagged "warning" or "fatal" is application-specific, so users

should familiarize themselves with the potential problems and decide for themselves on a project-by-project basis. The various possible alerts are described in Appendix A.

☞ **Beware OTD navigation.** As mentioned previously, the Microlab-1 satellite on which OTD was hosted frequently reported questionable navigation (attitude and ephemeris) data. This resulted in times in which the sensor spatial accuracy could be as low as 100-200 km. While fine for climatological work, this could be problematic in individual storm analysis. While work continues at the LIS/SCF to identify and repair bad Microlab navigation data, users should in the interim exercise caution when using OTD data for storm-scale case studies. If a storm's lightning signature looks "smeared", especially in comparison to other data sources, it is probably because of poor nav data. Note that LIS does not suffer from this problem.

☞ **Apples and oranges.** Be very careful when intercomparing OTD and LIS data. We have already mentioned that then sensors have different spatial resolution and sensitivity. These differences also have indirect effects, particularly on the algorithms used to assemble lightning groups into flashes and areas. Not only have the algorithms been refined to handle the increased number of events in the LIS data set, but the actual 'meaning' of the algorithms varies with the resolution and sensitivity of the sensors. Thus, flashes and areas reported by OTD may not correspond to the same physical entities as flashes and areas reported by LIS. Be cautious when trying to make direct comparisons - just because the data shares the same name, does not mean it is representing the same physical entity.

☞ **What is a flash?** All lightning detection and mapping sensors, be they RF or optical, must use some algorithm to cluster the individual measured components of lightning into the entity scientists term "flash". Physically, a lightning flash is of course simply a collection of contiguous channels which may be both conductive and radiant one or more times. However, actual lightning morphology is often too detailed for most instruments' resolution and accuracy, especially when the flashes and channels are concurrent in time or nearby in space. Thus, our assembly of optical pulses into the data element we call "flash" may be different than NLDN's or LDAR's assembly of RF bursts. Until the statistical relation of OTD and LIS "flashes" to other sensors' "flashes" is fully known, OTD- and LIS-derived flash rates should be used with caution, at least when comparing to other lightning data.

☞ **Consider regrouping.** The best way to ensure that the flash or thunderstorm area definitions are suitable for your specific scientific use would

be to devise your own "grouping" algorithm, which assembles LIS/OTD groups or events into "flashes" and "areas" of your own definition. All the necessary information for such a task is contained in the HDF/C/IDL data structures. While this obviously entails some programming on your part, you may easily come up with a much better technique for grouping the pulse data than we have. Note that you may also wish to regroup the data if you have filtered low-amplitude events in an effort to simulate a constant threshold setting in your data sample.

☞ **Be careful with radiance.** We report the total cloud-top radiance for each event, group, flash and area. This of course is the radiance as seen from space. If you plan to use these radiances scientifically, or try to convert them to optical energy, consider a few issues: the reported radiances are only for the range of our narrowband filter, bandwidths can be found in Koshak (); the radiances are not channel source radiances, but arise from multiple scatterings within the cloud; the optical depth of clouds may vary significantly; conversions to optical energy may require a plane-parallel assumption. While these issues do not preclude the scientific use of LIS/OTD-observed radiances, they may complicate it. Note also that the LIS sensor is slightly more likely to saturate than OTD; this was an intentional tradeoff to improve the sensor's resolution of low radiance values.

These are the major issues to consider when using LIS/OTD data. They may sound complicated, but are actually fairly easy to deal with once you familiarize yourself with the datasets. The best way to do this is to dig in and look at some data, before undertaking a rigorous analysis project. Fortunately, we've given you tools which will make this easy. The next chapter describes installation of the LIS/OTD software suite, and Part II of this guide will show you how to use it.

Chapter 3

Installation

This chapter describes the basic system requirements, installation and operation procedures for the LIS/OTD software package. Some unsupported platform configurations (i.e., certain Un*x variants) may require assistance from your local system administrator.

3.1 Macintosh PowerPC

Macintosh PowerPC support is available for all components of the LIS/OTD software package.

3.1.1 System Requirements

We recommend a minimum system of a 100 MHz PowerPC Macintosh or Macintosh clone, with a minimum of 48 Mb RAM and MacOS 7.5.x or higher. There is no Mac 68k support planned. Obviously a faster processor and more memory will dramatically improve the software performance. To use the IDL API and *LISAPP* program, you will need a licensed copy of *IDL 5.0.3*, available from RSI (`http://www.rsiinc.com`). *LISAPP* requires at least 1024x768 monitor resolution. It can be run at any color depth.

To use the C language API, you will need the *CodeWarrior Pro R3* compiler suite (the Academic version should work fine as well) from Metrowerks:

`http://www.metrowerks.com`

and the *HDF 4.1r1* distribution (full source tree) from NCSA:

`http://hdf.ncsa.uiuc.edu`

If you plan to keep the LIS/OTD datasets on a Un*x host, you might also consider the commercial *NFS/Share* package which allows Macs to NFS (network) mount Un*x disk volumes. The LIS/OTD software supports NFS mounted volumes as well as volumes mounted via AppleShare (e.g., if you keep your dataset on a WindowsNT or other Macintosh server). You will need the shareware program *Stuffit Expander* to unpack the software distribution.

3.1.2 Installation

Drag the file `LISOTD.sit.hqx` onto the *Stuffit Expander* icon to unpack the software distribution. This will create a folder named `LISOTD`, containing the folders `IDL`, `src` and `Documentation`.

If you have IDL, you'll need to first increase its minimum and preferred memory settings. Find the original IDL application (not its alias), select it, and choose "Get Info" from the File Menu. The minimum memory should be set to no less than 23000, and the preferred memory anywhere from 23000-64000, the more the better. The simplest way to use the IDL components is to drag all the contents of the `IDL` folder into your IDL startup folder; this method prevents you from having to reset any paths. If you wish to keep the IDL software separate, you must add the `LISOTD:IDL` path to your IDL paths via the IDL preferences menu option. You will also need to type '`cd,"<full path to LISOTD:IDL>"`' at the IDL command prompt each time you start IDL, or else configure your IDL startup.pro file to do this. To start LISAPP, type `.compile lisapp` and then `lisapp`. To include the LIS/OTD IDL API in your own programs, add `@lisotd.pro` and `@lisapp_DateTime.pro` at the top of your custom programs. Finally, if you wish to use the shared libraries in LISAPP or your own code to speed up file input (we strongly recommend this), be sure to choose the appropriate shared library. This is done in LISAPP by using the "Preferences...Input Settings" menu option, and in the API by specifying the shared library path and filename in your read routine. Chapter 4 describes the IDL software and the shared libraries in more detail.

To use the C API and libraries, open the project `UNIFIED` in the CodeWarrior Integrated Development Environment (IDE). We assume you know how to work within the IDE. You will likely need to add some paths to the project so it can find the HDF 4.1r1 libraries and include files (which we assume you have already installed elsewhere on your Mac). Choose the target "LISOTD Application". The file `reader.c` is the main code module of this target; it contains a basic program template which you can modify as you wish. If you want a console interface, say using the SIOUX package, you must of course add the appropriate libraries in CodeWarrior (refer to the CodeWarrior documentation for details).

Some further Mac tips: First, consider using the Motorola math libraries

libmoto. These are considerably faster than the fastest version of the bundled Apple math libraries, and are free from Motorola's web site:
`http://www.mot.com/SPS/PowerPC/library/fact_sheet/libmoto_download.html`
Second, if you are memory-limited and need to resort to virtual memory (using the Memory Control Panel), consider ugrading to MacOS 8.1. Virtual memory in System 8.1 is dramatically faster than it was in 7.5.x or 8.0, and has been reported to be even faster than the commercial *RamDoubler* product.

3.2 Windows 95/NT

Windows 95/NT support is available for the IDL components of the LIS/OTD software package. There is no Windows 3.1 support planned. Support for the C components may be available by Q3-Q4 1998.

3.2.1 System Requirements

We recommend a minimum system of a 100 MHz Pentium/AMD/Cyrix machine and 32 Mb of RAM. To use the IDL API and *LISAPP* program, you will need a licensed copy of *IDL 5.0.3*, available from RSI (`http://www.rsiinc.com`). *LISAPP* requires that you set your display to 256 colors before invoking IDL. It needs 1024x768 or higher monitor resolution to function correctly.

Although not yet available, initial C language support will eventually be with the *CodeWarrior Pro R3* compiler suite (the Academic version should work fine as well) from Metrowerks (`http://www.metrowerks.com`). Support for *Visual C++* from Microsoft (`http://www.microsoft.com`) may be added at a later date. The program *WinZip* or equivalent will be needed to unpack the distribution. Consult your system administrator for the appropriate software if you wish to access files stored remotely on a Un*x machine via NFS.

3.2.2 Installation

Open the file `LISOTD.zip`; this will create a folder named `LISOTD`, containing the folders `IDL` and `Documentation`.

The simplest way to use the IDL components is to copy all the contents of the `IDL` folder into your IDL startup folder. If you use a shared (networked) version of IDL, you may not be able to do this; in this case, you will need to add the appropriate paths to IDL in order to use this software. To start LISAPP, type `.compile lisapp` and then `lisapp`. To include the LIS/OTD IDL API in your own programs, add `@lisotd.pro` and `@lisapp_DateTime.pro` at the top of your custom programs. There is currently no shared library (DLL) support on Windows for improved file access speed; this support will be available once

the C APIs for Windows are released.

3.3 Un*x

Un*x support is available for the IDL components of the LIS/OTD software package, and the low-level C API. The high-level C API is only supported under Linux and IRIX, but may work on other systems. The shared libraries are only available under Linux and IRIX.

3.3.1 System Requirements

We recommend a minimum system of a 100 MHz Un*x box or Pentium/AMD/Cyrix system and 64 Mb of RAM. To use the IDL API and *LISAPP* program, you will need a licensed copy of *IDL 5.0.3*, available from RSI (`http://www.rsiinc.com`). To use the C APIs, you will need a C compiler. The commercial version available from your machine's manufacturer will probably give you the least trouble. The public domain `gcc` compiler will work fine on Linux boxes, but we cannot guarantee easy compatibility with `gcc` on other systems. You will also need the *HDF 4.1r1* distribution (full source tree) from NCSA (`http://hdf.ncsa.uiuc.edu`). You will likely need your system administrator to install C and HDF if they are not already available. Since the LIS/OTD datasets can be quite large, you might also want to NFS-mount various machines in your workstation cluster to conserve disk space; again, for this you must consult your sysadmin. Finally, you will need either the `uncompress`or `gunzip`commands available to unpack the software distribution.

3.3.2 Installation

First, uncompress the distribution file using:

`uncompress LISOTD.tar.Z` or `gunzip LISOTD.tar.gz`

depending on the archive you have downloaded. Next, unpack the tar archive file using `tar -xvf LISOTD.tar`. This will create a directory tree containing the directories `IDL`, `src` and `doc`.

To use the IDL *LISAPP* and API, just be sure to change directories to the `LISOTD/IDL` directory before invoking IDL or the IDL Development Environemnt (`idlde`). This way you shouldn't need to change any path settings, although you may want to if you move the software elsewhere. To start *LISAPP*, type `.compile lisapp` and then `lisapp`. To include the LIS/OTD IDL API in your own programs, add `@lisotd.pro` and `@lisapp_DateTime.pro` at the top of your custom programs. Finally, if you wish to use the shared libraries

in *LISAPP* or your own code to speed up file input (we strongly recommend this), be sure to choose the appropriate shared library. This is done in *LISAPP* by using the "Preferences...Input Settings" menu option, and in the API by specifying the shared library path and filename in your read routine. Chapter 4 describes the IDL software and the shared libraries in more detail.

To use the C APIs, a sample `Makefile` has been included in the `LISOTD/src` directory. You will need to adjust a few directory paths at the top of the Makefile; consult your system administrator for details. The file `reader.c` is a generic template you may use to write your own C applications. The following `make` options are available:

- `make clean` ... Remove all object code and libraries from this directory. Useful if things get mucked up or if you are upgrading or patching the software.

- `make allbutshared` ... Creates the basic libraries to be used by your application without attempting to create the C/IDL shared library. Also compiles the file `reader.c` to make sure it builds.

- `make all` ... As above but also attempts to compile the C/IDL shared library. The options in the distributed Makefile are only guaranteed to work under IRIX. For other platforms, consult your system administrator it the `make` fails. This option also compiles the file `reader.c` to make sure it builds.

- `make reader` ... The variant you will routinely use to compile your custom C code. This will compile or recompile the necessary libraries if they are not present or if their source code has been modified.

- `make reader_lowlevel` ... The variant you will use if you wish to only employ the low-level C API. This will compile or recompile the necessary library if it is not present or if the source code has been modified.

Adventurous users on non-Linux or non-IRIX platforms may want to try and recompile the C/IDL shared libraries to improve their IDL performance. The Makefile illustrates how the libraries are made on IRIX boxes, and might help you recompile. You'll likely need the help of your sysadmin; unfortunately, the LIS/SCF cannot provide assistance on unsupported Un*x systems. However, we'd certainly like to hear if you manage to get the libraries rebuilt, and to receive copies.

3.3.3 X Terminals / X Windows

When using Un*x-hosted IDL remotely via an X-terminal or X-window, there may be several additional considerations. For *LISAPP*, your monitor must be run at 1024x768 resolution or higher. Also, various X and X emulation software handle color table allocation differently. If you seem to be getting awkward colors in your *LISAPP* graphics windows, try restarting IDL, and typing `device,pseudo=8` before compiling and running *LISAPP*. If this works, you will need to either enter this at the start of each session or modify your IDL startup.pro file accordingly.

3.4 Other

IDL 5.0.3 is currently supported under OpenVMS. The LIS/OTD IDL API should function properly under OpenVMS. However, the *LISAPP* program would likely need some hacking in order to support the VMS directory/path syntax. Users interested in customizing the *LISAPP* code should contact the LIS/SCF before attempting this.

Currently, there is no support planned for OpenStep or BeOS. Limited C-language support for LinuxPPC/LinuxPMac/Mklinux may be available by Q3-Q4 1998. Apple Rhapsody for PowerPC and Rhapsody for Intel support is planned; the release date will be determined by the Rhapsody development schedule and timing of the initial HDF 4.1r1 port to Rhapsody.

Part II

High Level Interfaces

Chapter 4

IDL Interfaces: LISAPP

The *LISAPP* IDL program is a self-contained, GUI-based, menu (event) driven application which is invoked from within IDL. It allows direct input, analysis and visualization of LIS and OTD HDF data files, without any additional programming required by the end-user. It also leaves an inputted data file resident in IDL memory, and accesible as an IDL variable at the IDL command line. It is thus ideal for orbit-at-a-time data analysis. It is also easily extensible and modular, so users familiar with IDL widget programming can add their own functionality to the program.

4.1 General Usage

To use *LISAPP*, first start IDL or the IDL Development Environment. Depending on your platform, you may need to next set the working directory to the directory in which *LISAPP* is installed (see Chapter 3 for details). Next, type `.compile lisapp` and then `lisapp`. The main program window should appear, ready for use. If the window does not appear, and IDL complains about missing files, you haven't configured the startup directory properly.

Note that many of the menu options are initially "grayed out". *LISAPP* only activates menu options when it has enough data loaded to make them useful. You can read in a LIS or OTD data file by choosing **Open Orbit** from the **File** menu. Once an orbit is loaded, you're ready to go. Play around with some of the menu options to explore *LISAPP* further. The menu options are described in detail in section 4.4.

4.2 Recovering from Crashes

As with any developmental software package, *LISAPP* isn't completely bug-free. If you should happen to encounter a program-stopping bug (we've tried to eliminate as many as possible!), the following command sequence should restore full control of IDL, and restart *LISAPP*:

```
cd, "full path to the LISOTD/IDL directory"
widget_control, /reset
retall
xmanager
lisapp
; Be sure to use the correct directory delimiters
;   i.e., "/" on UNIX, ":" on Mac, "\" on Windows
; Example: cd, "/usr/people/me/LISOTD/IDL"
; Example: cd, "MacHardDrive:LISOTD:IDL"
; Example: cd, "C:\LISOTD\IDL"
```

Important note : We strongly recommend using only the "Done" or "Cancel" buttons available in most *LISAPP* windows to get rid of these windows, rather than the standard "close boxes" your computer may place on the windows. This will ensure that *LISAPP* does not try to manipulate windows which it is not aware have been closed (which will almost certainly cause a program crash). Future versions of the program will be more tolerant of close boxes.

One cause of program crashes at startup is bad a "preferences" file. *LISAPP* is able to be configured to remember certain user-specified options such as the location (and use) of shared libraries, default paths to your dataset, etc. It accomplishes this through a small "preferences file" created in your *LISAPP* directory. If you have copied the installation from another user, or your data directory disappears (is renamed, unmounted, etc.), *LISAPP* will fail to load the preferences file and crash. There is a simple solution to this. Start IDL, type `.compile lisapp` to compile the program, then `create_prefs`. This will create a new preferences file with generic "factory" settings. You only need do this once to repair a bad preferences file. Once you have done this, simply type `lisapp` at the IDL command prompt to begin working.

4.3 Command line interface

As mentioned above, once you have read an orbit into *LISAPP*, it is also available for inspection at the command line. At the command line, try typing:

```
print, orbit.point.lightning.flash(*).radiance
```

Neat, huh? Now try:

```
loadct, 39
window, 0, xsize=720, ysize=360, retain=2
map_set,/continents, xmargin=[0,0], ymargin=[0,0]
oplot, orbit.point.lightning.event(*).location(1), $
       orbit.point.lightning.event(*).location(0), $
       psym=4, color=64
```

As you can see, having the data resident in a single IDL variable can be *very* useful. Full details of the IDL "orbit" variable are provided in Chapter 5 and Appendix A.

4.4 Menu options

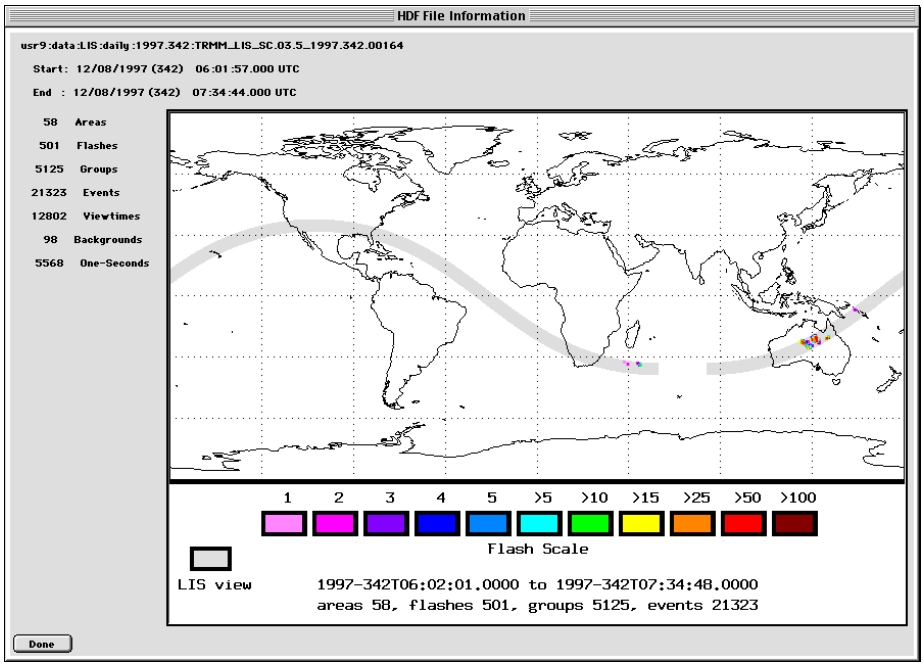
The *LISAPP* interface contains seven main pull-down menus, **File**, **Quick-View**, **Analysis**, **Export**, **Verify**, **Tools** and **Help**. Each controls a variety of relevant tasks *LISAPP* is capable of performing. This section describes the various *LISAPP* menu options in detail.

4.4.1 File menu

This menu contains all options related to HDF file input and output, and configuration of the Preferences file.

Get info...

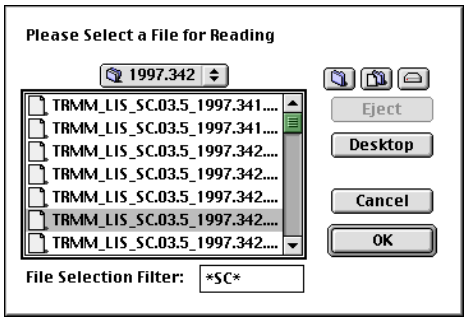
This option quickly loads and displays either a summary raster image (if available) or a summary table of the contents of a LIS or OTD HDF file. It is useful for rapidly isolating files with interesting data, before fully loading them into *LISAPP*.



Open orbit...

This option presents a dialog box which prompts the user to select a LIS or OTD HDF file for input. By default the box expects to read LIS data files,

for which it uses the character filter "SC" (which is embedded in all LIS HDF file names). If you wish to filter on OTD HDF file names, change the filter to "mlab*".



Note that reading OTD files will take slightly longer (and temporarily use more memory) as *LISAPP* must internally convert the old OTD data structure to a new LIS/OTD data structure for compatibility.

We recommend never renaming LIS or OTD HDF files, as these naming conventions allow the software to make initial guesses as to the type and content of the data files. If the file input seems painfully slow, you can either (a) tell *LISAPP* to use shared libraries, if these are supported on your system, or (b) tell *LISAPP* not to load individual data elements which may not be required, such as viewtime granules or events. Both of these options are invoked from the **File...Preferences...Input Settings** menu option. Note that with option (b), some menu options may not be available once the file is loaded. A third option is, of course, to run *LISAPP* on a faster computer!

Add more vdata

If you have chosen not to load certain data elements (such as viewtimes) but find in your analysis that you actually need them, you can use this option to add the needed elements into your loaded orbit. *LISAPP* will activate any additional menu options it can once the requested data have been loaded.

Save orbit as...

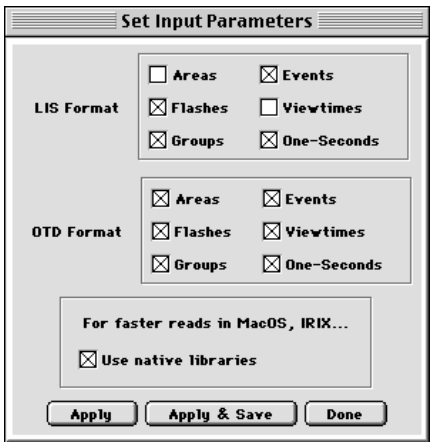
This menu option is currently active and working on some systems but unsupported. We do not recommend altering and resaving LIS/OTD HDF files at

this time.

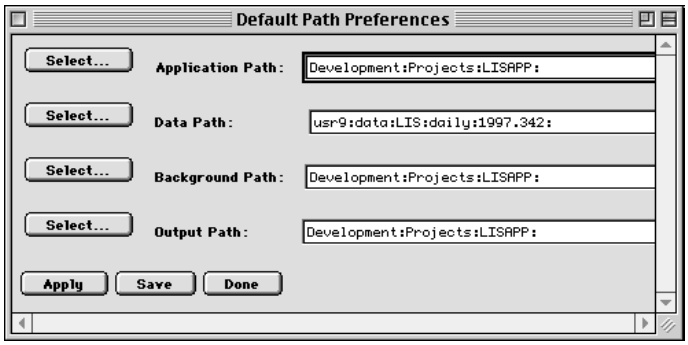
Preferences

Some *LISAPP* behaviors are configurable, and the changes are able to be remembered between IDL sessions. This is accomplished with a "preferences" file, which lives in the same directory as *LISAPP*. The options under this heading allow these preferences to be altered and/or saved. The preferences currently available for alteration are:

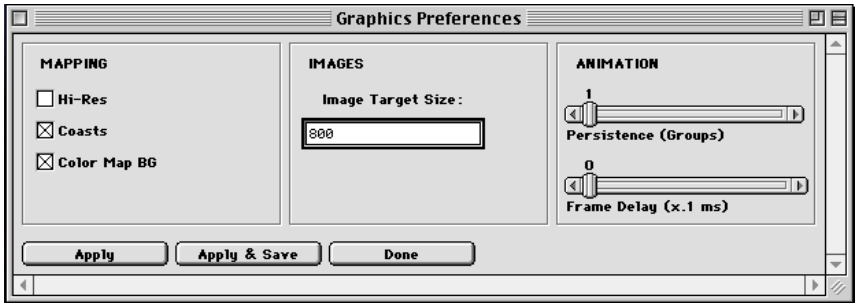
- **Input Settings** control which data elements are loaded when you read in a LIS or OTD orbit. If you are not using shared libraries, file input can sometimes be slow. If you do not need the viewtime, one-second or event data in your analysis, you might consider deselecting these elements in this control panel. This panel is also the location where you may activate use of the shared libraries (if your platform supports them).



- **Paths** control where *LISAPP* looks, by default, for LIS/OTD HDF data and other files. The "Data" path is the default path which will appear when the **Open orbit** option is first selected from the **File** menu. After the first file is loaded, *LISAPP* will default to whichever was the *most recent* directory used to load an HDF file. The "Background" path is similar to the "Data" path, but if you choose to keep background image HDF files in a separate directory, this allows you to specify that directory. The "Output" path is the default path for exported ASCII files, images and movie frames.



- **Graphics** controls several plotting and animation related parameters, such as whether to use hi-res maps, shaded continents and coastlines, how large to make certain plots, and how fast to run certain animations. Note that this panel may be used while other windows are open to interactively refine the appearance of some plots.



Should your preferences file ever become corrupted (e.g., if a default data disk goes away, etc.), you can restore the default settings by starting IDL and typing:

```
.compile lisapp
create_prefs
lisapp
```

Quit

This option quits *LISAPP* and frees all allocated memory.

4.4.2 QuickView menu

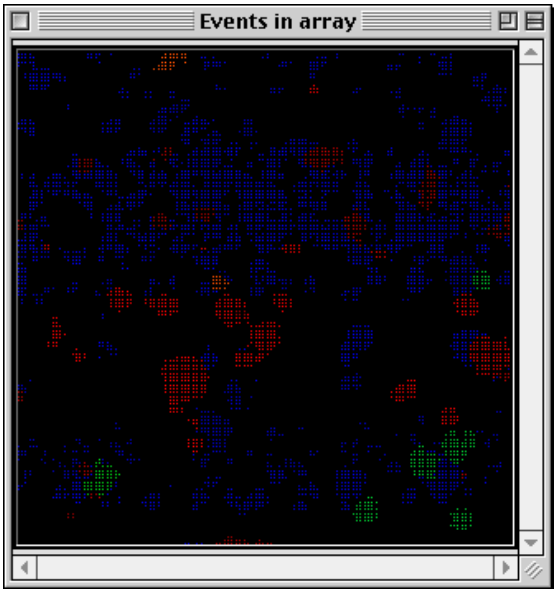
This menu contains options to quickly examine a file's contents to see if they are interesting enough for further analysis.

Browse image

This option displays either a raster summary image (if available) or a text summary table of the currently loaded file's contents. It is essentially the same as the **Get Info** option of the **File** menu.

Pixel array

This option generates a simple composite plot in *pixel coordinates* of all events observed during this orbit. This is a quick way of determining if there were high flash rate storms, large flashes, odd artifacts, etc. in the data file. The events are color coded by their relative time within the orbit (cool to warm color table indicating increasing time).



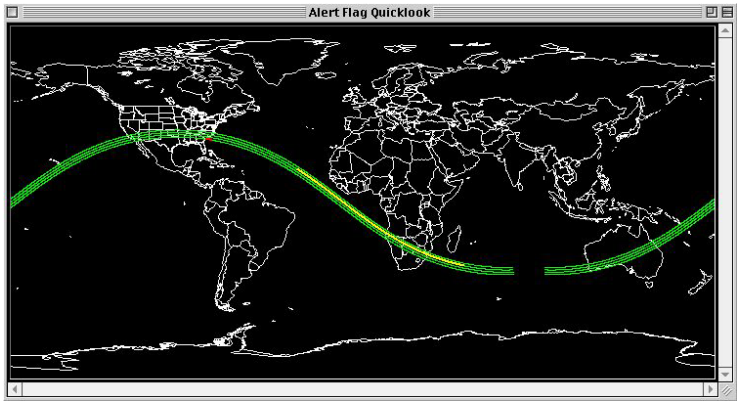
Backgrounds

This option generates a simple animation of all background images sampled during the currently loaded orbit. This option is a quick way to see if any interesting cloud systems such as hurricanes, frontal systems, MCS's, etc were in the orbit, as well as look for potential solar glint scenes.

Quicklook

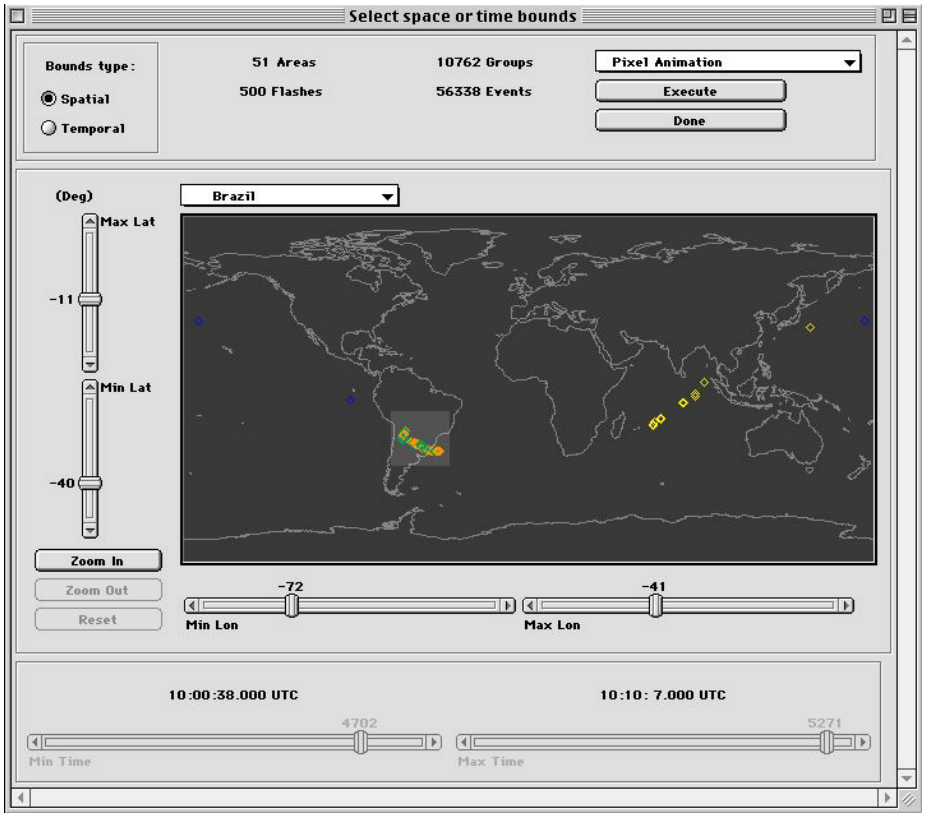
The options under this heading generate quick-and-dirty geographic plots of the areas, flashes, groups, events, viewtimes or background image locations contained in the current file.

An "alert summary" plot is also available showing the status (OK, warning or fatal) of each alert flag (instrument, platform, external and processing) at each second of the orbit. The quicklook plots are useful for searching for viewtime dropouts, problem regions during the orbit, areas of interest, etc. They are also useful in generating summary plots for OTD HDF files which lack embedded summary images.


Alert Flag Quicklook

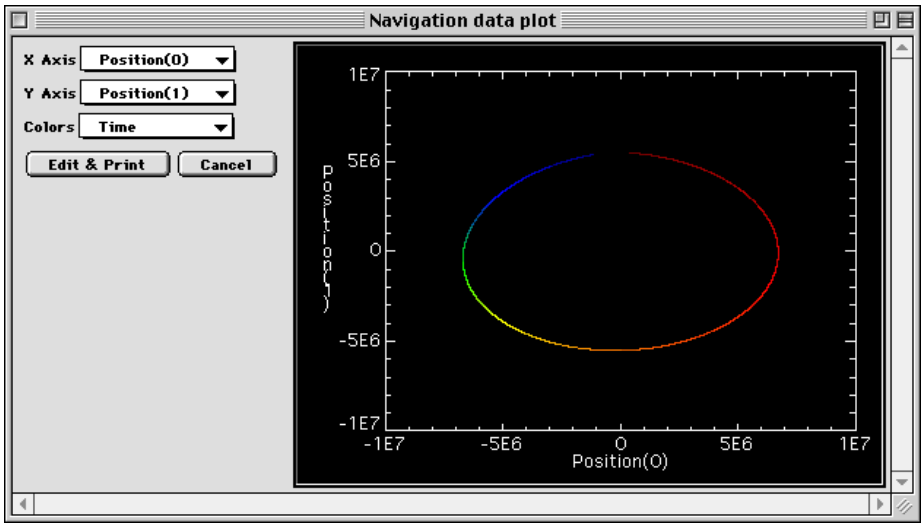
4.4.3 Analysis menu

This menu offers more advanced visualization and analysis tools. All routines in the **Analysis** and **Export** menus first invoke a "Bounds Selection" window, in which users may subset the orbit data either in space or in time. This tool then passes the subsetted data along to the requested **Analysis** or **Export** module.



Navigation data

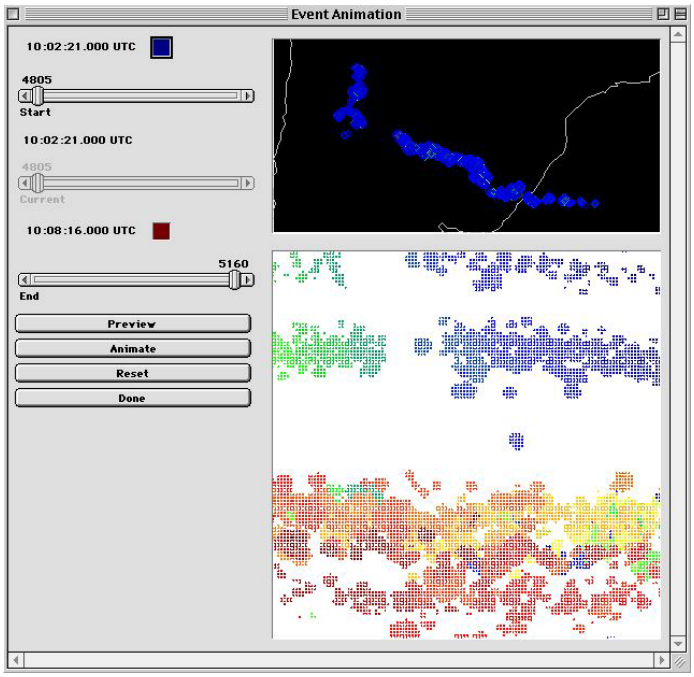
This option allows visualization of the sensor platform's (Microlab or TRMM) ephemeris and attitude data. This is especially useful for OTD data, if users suspect a period of poor navigation data is "smearing" the geolocated lightning positions.



Noisy navigation data is readily visible in these plots, as attitude, ephemeris, position and velocity should be more or less smoothly-varying over the course of a given orbit. Discrete jumps, high frequency noise, etc. all indicate poor quality navigation data. Note that the navigation data for LIS is usually excellent and does not require detailed verification.

Pixel animation

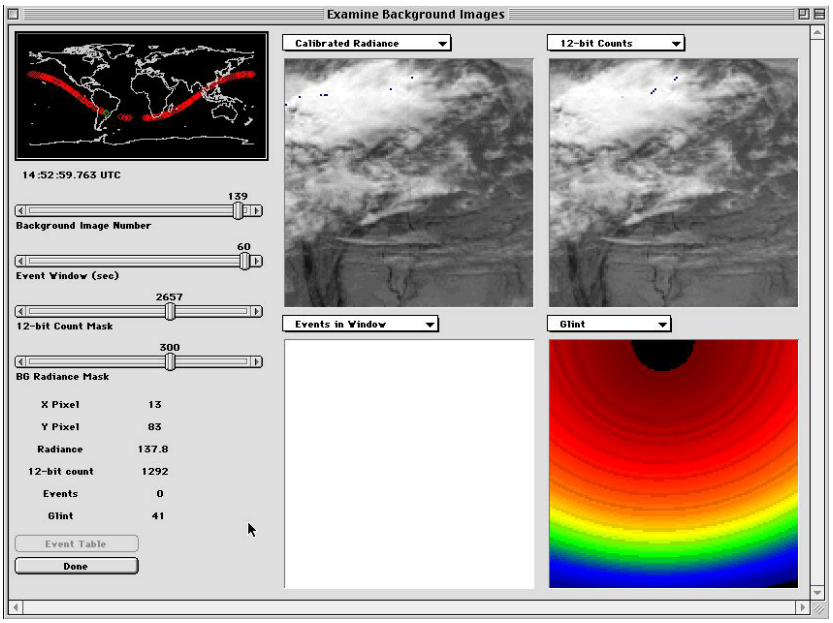
This option provides basic visualization of lightning flashes in animation mode. Since the animations can be quite lengthy and do not currently support interrupts, a subsetting tool (pair of slide-bars) is provided to narrow down the window chosen for animation. All lightning events in the subsetted window are first plotted in a world map window, color coded by increasing time. The sliders narrow the time window, with small color indicators showing the earliest and latest events in the map plot which will be animated. The "Preview" button zooms the map and shows only those events within the selected window. The "Animate" button begins the animation. The animation is sequential in *frames with groups*; i.e., each successive animation frame is the next 2 ms sensor observation frame in which a lightning group was observed. "Dead time" with no observed groups is not included in the animation, so this is not a "real-time" movie. The base frames for creating a "real time" movie may be generated using the **Make Movie Frames** module in the **Export** menu.



Note that preferences for this module may be set under the **File... Preferences... Graphics** menu option.

Thresholds and glint

This option provides a detailed interface for examination of the background radiance properties of a given storm scene. The module works in *pixel coordinates*. Four subwindows are provided: the background scene in raw amplitude counts, the background scene in calibrated radiance, an event window, and a multipurpose plot window. The background scene is selected with a slider bar. The event window is also controlled by a slider bar; this sets the time window around each background scene during which to consider events. Since background images are only saved every 30 seconds or so, and the array translates considerably during the inter-image time, keeping this event window set to 1-3 seconds will guarantee that any events plotted in this window correspond to the rendered scene. Setting the window to a longer interval may help identify scene-related noise anomalies such as solar glint, contrast effects, etc.



The remaining two slider bars control the "masking" applied to the background plots. Normally, the background scenes are plotted completely in greyscale. Any values above the thresholds selected with these sliders will, however, be plotted in color. This is a convenient method for isolating very bright backgrounds, or for gauging the maximum background radiance for a given scene.

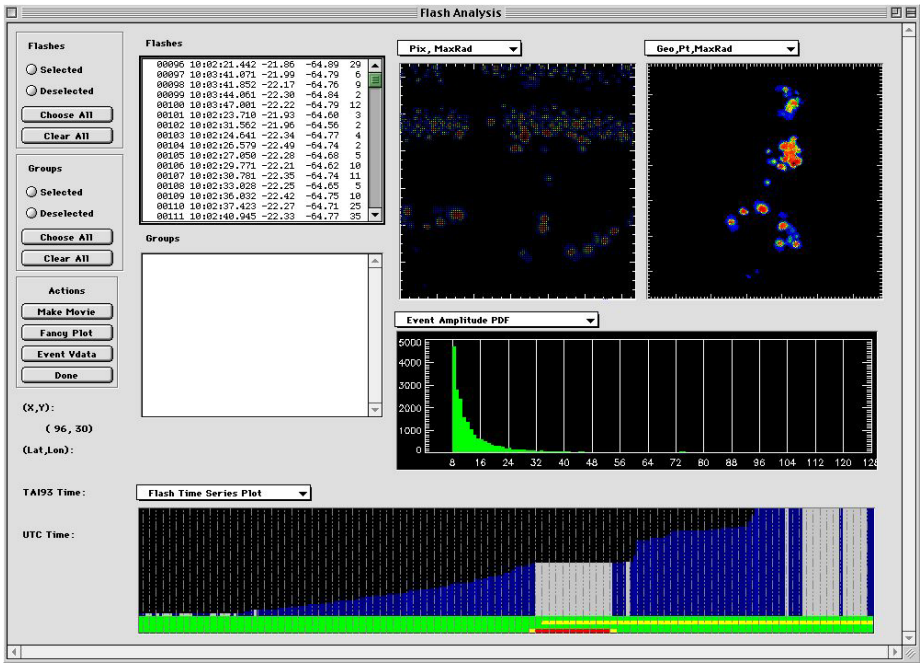
The multipurpose plot has three options. The first displays the (analytically derived) solar glint angle at the time of this background scene. The plot runs in a "warm-to-cool" color table, with "warmer" (redder) values indicating a higher likelihood of solar glint. Most solar glint is automatically removed from the dataset by the LIS/SCF production code, however the occasional anomaly

may still introduce some false events. Note that these may also occur at off-peak glint angles from 30-45 degrees; this may be due to a secondary ice scatterig peak. The second plot option displays the applied thresholds for this scene. With OTD, this plot is uniform, as the sensor does not support variable thresholding. With LIS, this will vary discretely with the background scene radiance. It may be useful to examine a storm of interest for threshold changes across the cloud if a detailed case study is being performed. The final plot option shows a histogram of all background radiances, and a histogram of the background values at which lightning events within the chosen time window occurred. In any given scene, most events *should* fall on the brightest (cloud) pixels. If they do not (i.e., if the event histogram is biased towards the low end of the background scene values), it may be an indication that false events are present.

Note also that each window is "active"; i.e., labels in the left-hand column update continuously as the cursor is moved across the various images. Finally, note that any events occurring in the chosen time window can be passed directly to the **Export...Vdata...Event** table view module.

Flash analysis

This option provides an interface for a detailed analysis of lightning flashes or storm regions. It is designed to operate on small geographic regions (not the entire orbit), so be sure to select a reasonably-sized geographic boundary in the subsetting window. The Flash Analysis window has several subcomponents. On the left is a list of flashes (and eventually groups) which fall within the selected spatial bounds. On the right and bottom are several plot windows. From bottom to top, these include a "time series" plot, an "event histogram" plot, a "pixel space" plot and a "geographic plot" of the subsetted data.



The "time series" plot shows the occurrence of lightning groups within the chosen flashes, indicated as vertical, colored bars (gray bars denote "dead time" with no observed groups). The y-axis in this plot represents cumulative radiance for the subsetted data, so the plot may be used to isolate very bright groups and flashes (rapid jumps in y). The fiduciary marks are either dotted (100 ms intervals) or dot-dashed (1 sec intervals); the 100 ms tics are dropped if the time window chosen is very long. Running horizontally across this plot are four bars indicating the status (OK, warning, fatal) of the four alert flags (platform, instrument, external, processing), for quick identification of potential hazards and viewtime dropouts.

The "event histogram" by default shows a histogram of all event raw amplitudes contained in the subsetted flash data. For real lightning, this plot should

look roughly lognormal; other distributions suggest false data which was missed by our production code filters. Note that with LIS data, a small "hump" near 75-80 counts is normal; this is a result of our piecewise-linear calibration (gain) designed into the sensor. Several other plots are available from the pulldown menu to view the event distribution as various combinations of log and linear PDFs and CDFs.

The "pixel space" plot shows all subsetted events plotted in pixel array space. The events are color coded and sorted in amplitude, so the plot shows the brightest event at each *(x,y)* location in the array. The "geographic plot" is very similar but plots in actual (geolocated or lat/lon) coordinates. This plot has three suboptions; a quick-plot (1 point per event) mode, a footprint mode (each actual pixel polygon is filled in), and a footprint+background scene mode (same as footprint but with the background scenes overlaid). Note that each successive option is increasingly more CPU-intensive; the 'footprint+background' option should only be used once the data has been subsetted to the desired level.

Note that the flash and group lists are interactive; users can user either the toggle buttons or can double click on the list entries to select and deselect individual flashes and groups in the list (the window initially plots with everything selected, then reverts to "all deselected" to allow interactive subsetting).

Note that the time-series plot may be very useful for isolating "FIFO full", or "sensor blinding" conditions which users may have missed by ignoring the lightning data or one-second alert flags. Very active storms with unusual 10 second dropouts are quite likely storms in which the amount of optical pulse (event) data flooded the sensor FIFO buffers, resulting in a loss of viewtime. Again, this may be confirmed by examining the alert flags available in the lightning point data and one-second data elements. Remember - viewtime cannot be assumed constant, and must be verified!

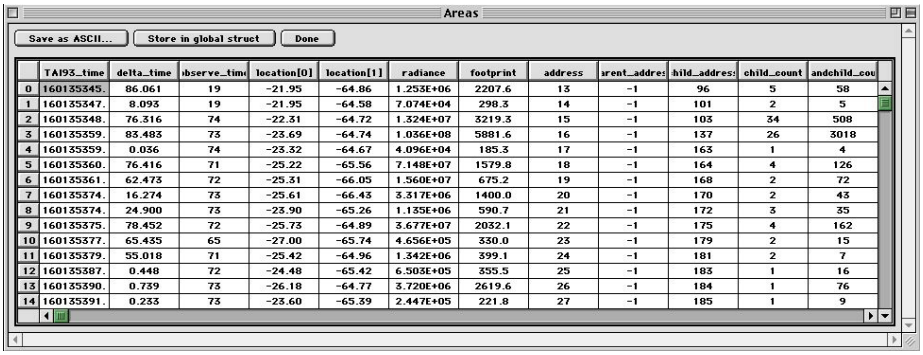
Finally, note that all subsetted lightning event data may be passed directly to the **Export...Vdata...Event** table view module for more detailed analysis.

4.4.4 Export menu

This menu contains options to examine and visualize the HDF file contents in more detail, as well as export data and visualizations to external files.

Vdatas

The menu options under this heading allow users to view and export tabular summaries of some of the data elements in the HDF file, including areas, flashes, groups, events, viewtimes and one-second data:



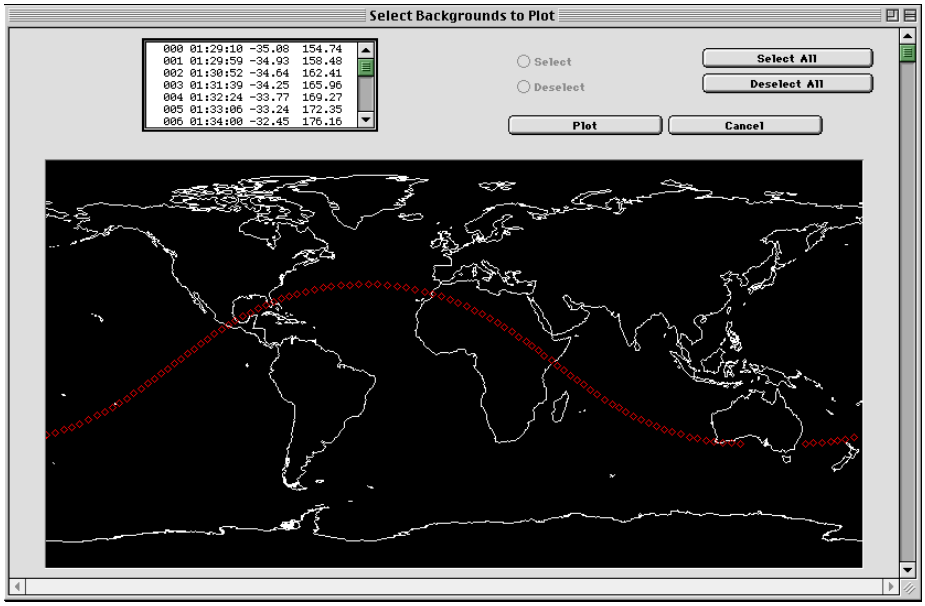
	TAI93_time	delta_time	observe_time	location[0]	location[1]	radiance	footprint	address	parent_addres	child_address	child_count	andchild_cou
0	160135345.	86.061	19	-21.95	-64.86	1.253E+06	2207.6	13	-1	96	5	58
1	160135347.	8.093	19	-21.95	-64.58	7.074E+04	298.3	14	-1	101	2	5
2	160135348.	76.316	74	-22.31	-64.72	1.324E+07	3219.3	15	-1	103	34	508
3	160135359.	83.483	73	-23.69	-64.74	1.036E+08	5881.6	16	-1	137	26	3018
4	160135359.	0.036	74	-23.32	-64.67	4.096E+04	185.3	17	-1	163	1	4
5	160135360.	76.416	71	-25.22	-65.56	7.148E+07	1579.8	18	-1	164	4	126
6	160135361.	62.473	72	-25.31	-66.05	1.560E+07	675.2	19	-1	168	2	72
7	160135374.	16.274	73	-25.61	-66.43	3.317E+06	1400.0	20	-1	170	2	43
8	160135374.	24.900	73	-23.90	-65.26	1.135E+06	590.7	21	-1	172	3	35
9	160135375.	78.452	72	-25.73	-64.89	3.677E+07	2032.1	22	-1	175	4	162
10	160135377.	65.435	65	-27.00	-65.74	4.656E+05	330.0	23	-1	179	2	15
11	160135379.	55.018	71	-25.42	-64.96	1.342E+06	399.1	24	-1	181	2	7
12	160135387.	0.448	72	-24.48	-65.42	6.503E+05	355.5	25	-1	183	1	16
13	160135390.	0.739	73	-26.18	-64.77	3.720E+06	2619.6	26	-1	184	1	76
14	160135391.	0.233	73	-23.60	-65.39	2.447E+05	221.8	27	-1	185	1	9

Note that since these tables may be very large (and IDL must format them), they may take a while to appear. Be patient ... unless it tells you so, IDL has not crashed, it is merely formatting lots of data. Also note that on some desktop (PC or Mac) systems with limited memory, these options may consume too much memory and cause the application to crash. We recommend only using these options if you have adequate memory on your desktop computer system (see Chapter 3 for recommended values).

Each table window contains the buttons **Save as ASCII...** and **Save in global struct**. The former button exports the table data into an ASCII file of your choosing, formatted as in the table window. The latter button stores the subsetted data visible in the window in an IDL global structure variable, named `subset_areas`, `subset_viewtimes`, `subset_one_seconds`, etc. This is useful if you have used the bounds selection tool to subset your data, and now wish to use the IDL command line to quickly calculate some statistics, such as extrema, or quickly create your own plots with the subsetted data. Details on the IDL LIS/OTD structure variables can be found in Chapter 5.

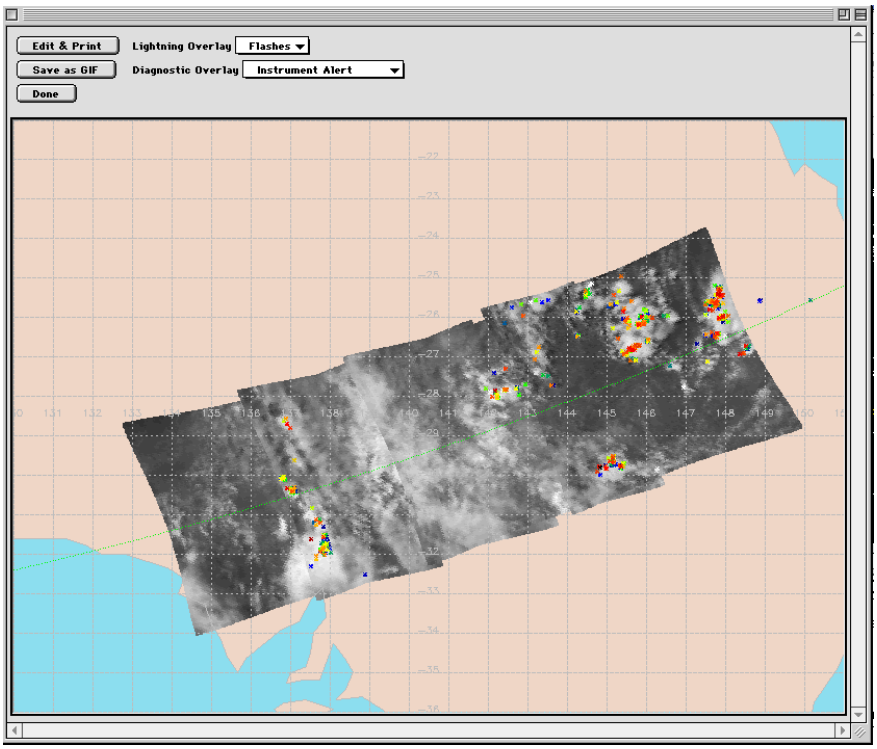
Location plot

This option allows a custom geographic plot to be generated with map overlays, background images, lightning overlays, and platform summary information all on the same plot. Once apporpriate bounds have been selected, this option creates a second "selection" window in which users may further refine which background images / geographic regions to plot:



The scrolling window lists the times and nadir points of the available background images. Initially, only those backgrounds falling within your chosen bounds are highlighted in the large map window. You may select (or unselect) additional backgrounds from the list.

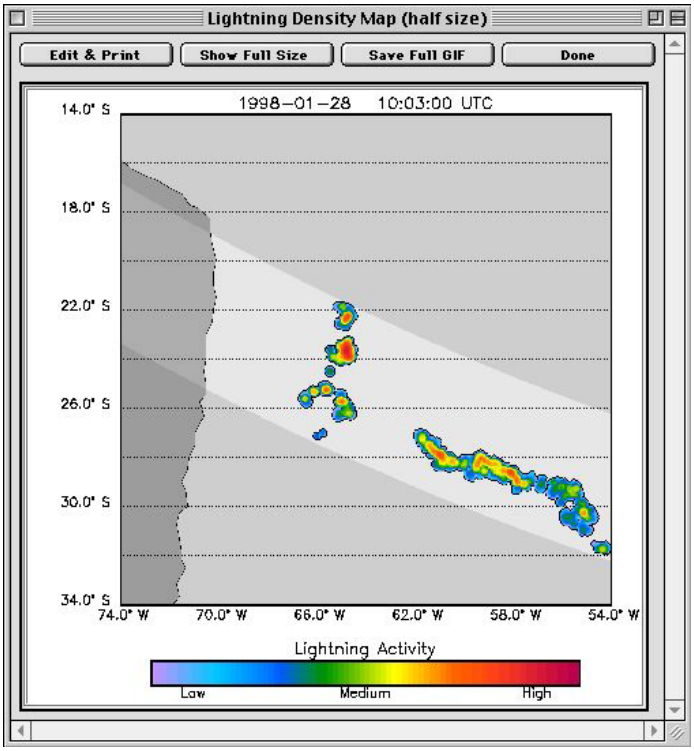
Once selected, a "base" plot is generated and lightning and platform data overlays are interactively available. A final "save as GIF" option is available, as well as a slightly slower "Edit and Print" option which uses IDL's built-in object tools to send the image to a printer.



Note that the settings in the **Edit-Preferences-Graphics** menu option control the appearance of the resultant plots, such as size, map overlay, etc. These may be changed when the selection window is open

Density map

This option creates a presentation-quality lightning "density map" with a field-of-view and geographic overlay. The map spans twenty degrees of latitude and longitude for LIS, thirty degrees for OTD. It will be centered based upon the geographic region you have chosen in the bounds selection window.

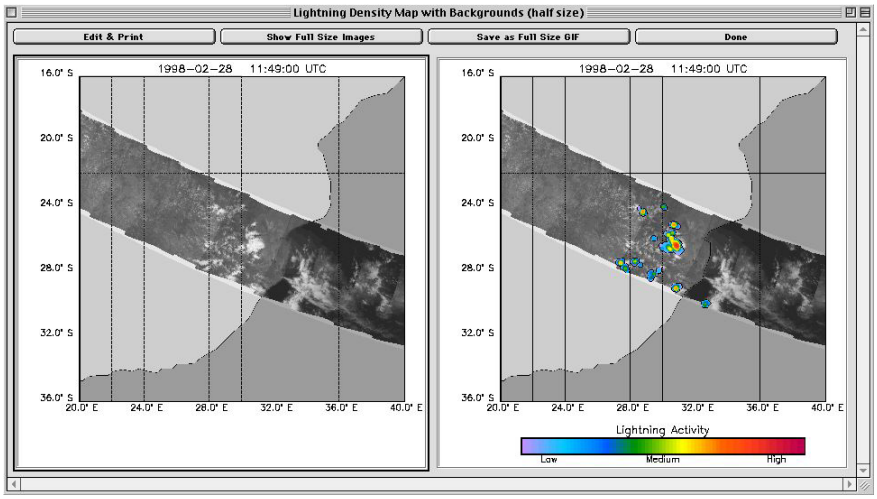


The map renders total *group* density on a log scale, and is adjusted for viewtime (i.e., viewtime dropouts due to sensor buffer overflows are considered). At this time, it is presented on a "relative" (low, medium,high) scale; as we become more confident of the LIS calibration and sensitivity, we will eventually change this to an "absolute" scale.

The map is shown at half-size in the default window; users with large monitors can use the **Show Full Size** button to view it at full scale. The **Edit and Print** and **Save Full GIF** buttons offer two ways of exporting the rendered image. Support for other output graphic formats may be added at a later date.

Density map with backgrounds

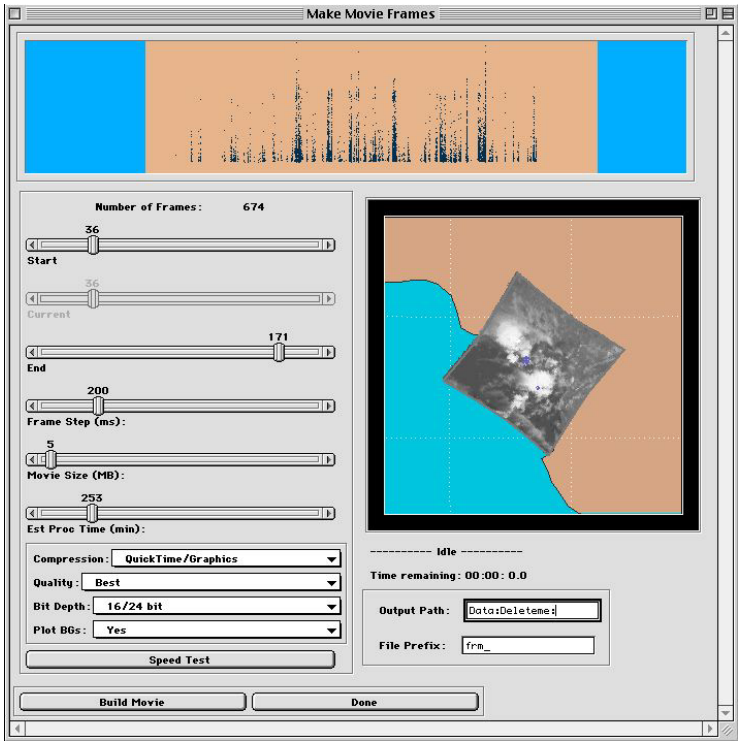
This option is very similar to the **Density Map** described above, but adds overlays of the geolocated background images. This option is somewhat more time-consuming but yields the best quality presentation plots.



Note that two images are created; one with only the background scenes and one with both backgrounds and the lightning density. This may be useful if the lightning hides too much of the background scene detail. The **Edit and Print**, **Show Full Size** and **Save Full GIF** buttons function as described above.

Make movie frames

This option allows you to generate a series of GIF images which can be later composited into an animation of a specific storm overpass. A number of configuration controls help you constrain the size, frame rate and time-to-process of the resulting sequence. Note that this module does *not* actually create the final movie; it merely generates the frames. You are responsible for finding a suitable third-party application to composite the frames into a single movie.



The window for this module is shown above. A simple "time series" plot of event amplitudes is found at the top of the window. By default, the bounds selection tool passes a short "lead in" and "lead out" time window to this module to improve the movie quality, thus the "dead time" usually found at the beginning and end of this plot. Beneath this are a number of configuration sliders and pull-down menus on the left; to the right is the main rendering window and a status bar.

The configuration sliders are interlinked to enable you to tune your movie generation to meet a particular constraint, be it movie size, time-to-process, frame rate, etc. For example, changing the start and stop times updates the **Number of frames** label, as well as the **Movie Size** and **Estimated Processing Time** sliders. The options, in detail, are as follows:

- **Start/Current/End** These sliders control the relative start and stop times of your movie. The units are 1-second intervals; the selected interval is highlighted in the time-series plot at the top of the window. Reducing the movie interval of course will help you minimize your movie size and time-to-process.

- **Frame Step (ms)** This is the interval at which movie frames will be generated, in milliseconds. For example, a 200 ms (the default) interval plots all events occurring in each 200 ms time step. Reducing this will obviously cause your movie size and time-to-process to grow, but give you more interesting animations. If you want your final movie to run at a known fraction of real-time, remember this setting and, in your third-party movie generation software, configure the desired frames-per-second setting accordingly (e.g., 5 fps would be real-time for a 200 ms time step).

- **Movie Size** This is the estimated movie size, in megabytes, based on your selected rendering window, frame step, compression settings (see below), and choice of whether or not to plot background images. The value is based on typical scenes, and probably valid to within about 10%.

- **Estimated Processing Time** This is the estimated time to finish rendering and saving all the movie frames, based on your selected time window and frame interval. This is of course highly dependent on your CPU speed and load. The bar is initially greyed out, with "default" settings based on a 200 MHz PowerPC machine. The "Speed Test" button performs a short run-through of about a dozen frames to try and gauge what the actual processing time will be on your system. Once the speed test has been performed, the **Est Proc Time** slider will be enabled and you can use it to further constrain your movie settings. Note that this estimate is probably only good to about 25%, as the actual processing time is somewhat dependent on the amount of lightning data actually in your movie.

- **Compression, Quality, Bit Depth** These pull-down menus don't actually *set* any of these options in the rendered frames; rather, they are used to describe the eventual characteristics of your final movie, in order to estimate the final movie size. You are responsible for configuring your third-party movie generation utilities to use the options you specify here.

- **Plot BGs** This option toggles the plotting of background images in your movie frames. Note that this is *very* CPU-intensive and will greatly lengthen your total processing time (as well as increase the final movie

size somewhat). Be sure that your scene is actually in daylight and that you definitely want background images before selecting this option.

- **Status, Time Remaining** These labels show the current module status and estimated time remaining (in HH:MM:SS) for the current speed test or movie frame build. Note that the **Time Remaining** information is probably not terribly accurate until about a quarter to a third of the way through the build.

- **Output Path, File Prefix** These specify the output directory (path) and file prefix of your movie frames. For example, specifying the path `/usr/me/` and prefix `frm_` will result in files named `/usr/me/frm_0000.gif`, `/usr/me/frm_0001.gif`, etc. The numerically-increasing file names are a fairly standard convention used by image compositing software. Some notes: you *must* have write-permission to the output directory, and the entry here *must* contain a directory separator as its last character (e.g., '/' on Un*x, ':' on MacOS, 'ón Windows). An easy way to set this is to change the output file path in the **File... Preferences... Paths** window.

This tool works well for both LIS and OTD files. The image rendering has been optimized as much as possible under IDL; we apologize for the long processing time but there's not much more we can do to speed it up.

4.4.5 Verify menu

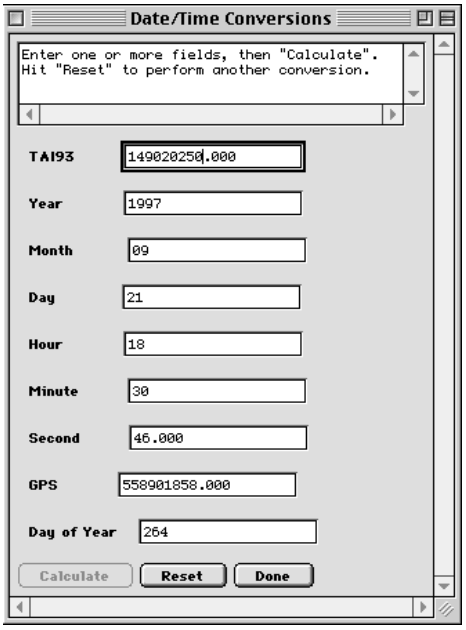
This menu is disabled in the end-user distribution of *LISAPP*. It contains diagnostic tools primarily used by the LIS/SCF staff to guarantee data file integrity, and should not be relevant to end-users.

4.4.6 Tools menu

This menu provides several utilities which may be helpful to end-users in their analysis of LIS/OTD data.

Date/Time conversions

This is a simple module for converting between the UTC, GPS and TAI93 time systems. The module is fairly flexible, and will attempt to populate as many missing fields as it can with the information it is given. See Chapter X for an IDL procedure which performs these functions and can be invoked from the command-line or from within a user's IDL program.



Alert flag decoder

This is a simple calculator which translate bit-masked alert flag summaries and alert flags in lightning point data and one-second data into human-language

alert descriptions. See also the options under the **Help** menu.

This tool is not available in the initial (prerelease) version of the LIS/OTD software suite.

4.4.7 Help menu

This menu will eventually contain detailed online help for *LISAPP*.

Flags

This option displays tables containing the bit interpretations for the various bit-masked alert summary flags and alert flags in the LIS/OTD lightning point data and one-second data.

4.5 Adding modules

Although the *LISAPP* IDL code is quite lenghty, it is actually fairly easy to add custom modules to the program. A basic knowledge of IDL widgets and IDL event-driven programming is, of course, required. Beyond that, developers need only be familiar with the IDL-flavord LIS/OTD data structure, and any necessary IDL analysis, plotting or data export routines needed for the custom data module.

4.5.1 Widget interface

Basically, module developers need only worry about six steps in their modification of the *LISAPP* code:

1. Create your own common block containing the menu IDs of the modules you wish to add to *LISAPP* (see item 2, below). These common blocks will need to be added to the procedures small `lisapp`, `lisapp_menu_event`and `new_sensitize_menus`.

2. Create the new menu buttons in the main *LISAPP* window. Templates for menu items can found in the main widget creation routine in the `lisapp` procedure.

3. Add a trap for your procedure in the *LISAPP* event handler (a simple addition to the main `case` statement in the event handler which invokes your module when the appropriate menu item is selected).

4. Write your module! The `datafiles` common block can be used to access the loaded orbit data. The `lisbase` common block provides the widget id of the main *LISAPP* window, which should be the group leader for your new widgets (if your module creates a new widget hierarchy; if it is just a plot or export routine, this is not needed). The `sensor_read` common block reports which sensor (LIS or OTD) and which file format (OTD or LIS/OTD) corresponds with the loaded data, if this information is needed. Finally, the `lisapp_colors` common block contains the main color table used in the program (see below).

5. Determine which data (areas, flashes, groups, events, one-seconds, view-times, background image summaries and/or background images) are used by your module. Modify the procedure `new_sensitize_menus` to sensitize or desensitize your menu option, depending on what subset of LIS/OTD data has been read in from an HDF file. Since users have the option of

partially loading HDF files to save time, this procedure is used to decide which menu options are relevant for each subset of data. Details on how to do this are found in the comments within the `new_sensitize_menus` procedure.

That's it! It should be fairly straightforward if you follow the outline above. The LIS/SCF can help developers with specific details of the *LISAPP* code, but cannot provide tutorials on basic IDL widget/event programming - please consult the IDL online or printed documentation for the latter.

4.5.2 Color tables

Handling color in IDL is not a trivial matter. Remember that at its best, *LISAPP* should be fully cross-platform deployable. Effectively, this means that we have to live with the lowest-common denominator graphics device, which today means 256 colors. Further, since IDL is often run over X windows, even fewer than 256 colors are likely to be available. We have found that color tables with 220 elements should be adequate for most of today's IDL environments.

As shipped, *LISAPP* uses two different color tables. The default table contains 63 grayscale values, followed by 63 "rainbow values" and a few special-purpose colors (this table is accessed via the `lisapp_colors` common block. A second table is used in the **Density Map** and **Make Movie Frames** modules; this contains 220 colors including some "land" gradations, "ocean" gradations, "rainbow" values and grayscale values.

You are free to use any 220-or-fewer length color table you can think of, however, we strongly suggest that your module store the previous color table before it begins (via `tvlct,oldr,oldg,oldb,/get`) and restore it when it is finished (via `tvlct,oldr,oldg,oldb`). Note that this of course means your module must be widget-based and have an event-handler which traps for a "done" or "cancel" condition. If your module simply creates a new plot window and doesn't include widgets and event-handling, we can't guarantee the rest of *LISAPP* will function properly (color-wise) after your module is called. This requirement will be relaxed over time as we work to ensure that all of *LISAPP*'s modules handle all their colors internally.

4.6 Future plans

Future plans for *LISAPP* functionality *may* include:

- More sophisticated scene animation

- Extension of the **Flash Analysis** module to include areas and groups
- More sophisticated (formal-looking) plot formats
- Shared library (DLL) support for Windows95/NT
- Import and concurrent rendering of other (NLDN, NEXRAD, GOES, TRMM) data

While there is no specific timetable for these improvements, most should be available by Q3-Q4 1998.

Chapter 5

IDL Interfaces: API

The IDL language high-level LIS/OTD API consists of several main components: a set of IDL structure variable definitions to hold LIS/OTD data, a simple procedure for LIS/OTD HDF file access and several useful utility routines. Together, they should facilitate rapid development of custom IDL analysis and visualization code by LIS/OTD end-users. To use the API, familiarity with the basic LIS/OTD data structure (Chapter X) and the IDL programming language is required.

5.1 Data Structures

Structure variables in IDL are quite similar to structure variables in C, at least in terms of their calling reference and syntax. Users not familiar with IDL structures should not be intimidated; all the details of structure declaration and allocation are handled transparently by the LIS/OTD API; users need only know how to reference the variables. As in C, structures may be nested (i.e., one may create "structures of structures"); we utilize this feature to develop IDL structures which almost exactly parallel the data organization in the HDF files and the C high-level API data structures.

Users need only be familiar with three data structures:

- An "orbit" data structure (an anonymous IDL structure, for those interested) which contains the science data contents of an entire HDF file. The science data (typically 1-3 Mb per file) is loaded using the `READ_ORBIT` procedure (see below), and items referenced directly, e.g.:

```
print, orbit.one_second[*].alert_summary

x = moment(orbit.point.lightning.flash[*].radiance)
```

```
plot, orbit.point.lightning.event[*].location[1], $
      orbit.point.lightning.event[*].location[0]
```

The orbit data structure mirrors the generic LIS/OTD data hierarchy, i.e.:

```
someOrbit.orbit_summary.*
someOrbit.one_second[].*
someOrbit.point.*
    someOrbit.point.point_summary.*
    someOrbit.point.viewtime[].*
    someOrbit.point.bg_summary[].*
    someOrbit.point.lightning.*
        someOrbit.point.lightning.area[].*
        someOrbit.point.lightning.flash[].*
        someOrbit.point.lightning.group[].*
        someOrbit.point.lightning.event[].*
```

The complete data elements for each substructure (e.g., radiance, location, etc) are summarized in Appendix X.

- A "UTC" data structure containing the subfields of a UTC date/time identifier:

```
someUTC.year
someUTC.month
someUTC.day
someUTC.hour
someUTC.minute
someUTC.second
someUTC.doy
```

(Note `doy` is the "day of year", useful in sequential ordering applications).

- a "DateTime" data structure containing various notations for the same date/time:

```
someDateTime.TAI93
someDateTime.GPS
someDateTime.UTC.*
someDateTime.Local.*
```

(Note `UTC` and `Local` are "UTC" format structures; see above).

5.2 Interface Routines

The primary interface routine in the IDL API are the `READ_ORBIT` and `READ_OLD_ORBIT` procedures. These load LIS/OTD HDF data files into the "Orbit" data structure described above. The calling interfaces are described below.

5.2.1 READ_ORBIT

The READ_ORBIT procedure loads an entire orbit's science data from a LIS/OTD (new format) HDF file. The data is returned in a single IDL "structure of structures" variable (see above). A dialog window monitors the progress of the file input.

Calling Sequence

READ_ORBIT, *FILENAME*, *ORBIT*

Arguments

FILENAME
The HDF data file *full* file name (i.e., path + name).

ORBIT
A named variable in which the orbit science data will be returned. This need not be declared beforehand.

Keywords

NATIVE
Set this keyword to the full path and name of the shared libraries to be used to accelerate file input, if these are available on your platform.

QUIET
Set this keyword to suppress the status window updates during the orbit load; this is useful for background (batch) processing.

NO_VIEWTIMES
Set this keyword to neglect input of viewtime granules, if shared libraries are not available for your platform, viewtime data is not needed (i.e., lightning *rate* computations are not being performed) and you wish to reduce the file input time.

NO_ONE_SECONDS
Set this keyword to neglect input of one-second data, if shared libraries are not available for your platform, one-second data is not needed (i.e., you do not wish to check alert flags or plot geolocated background images) and you wish to reduce the file input time.

NO_AREAS
NO_FLASHES
NO_GROUPS
NO_EVENTS
Set one or more of these keywords to neglect input of the associated lightning point data, if shared libraries are not available for your platform, the specific point data is not needed and you wish to reduce the file input time.

Examples

Load an orbit using shared libraries. Suppress the status window:

```
shlib = $
  "/usr/me/LISOTD/IDL/LISOTD_Irix.so"
fn    = $
  "/data/1998.070/TRMM_LIS_SC.03.5_1998.070.1640"
READ_ORBIT, fn, orbit, NATIVE=shlib, /QUIET
```

Load an orbit using native IDL libraries, neglecting viewtimes to speed things up:

```
fn = $
   "/data/1998.070/TRMM_LIS_SC.03.5_1998.070.1640"
READ_ORBIT, fn, orbit, /NO_VIEWTIMES
```

5.2.2 READ_OLD_ORBIT

The READ_OLD_ORBIT procedure loads an entire orbit's science data from an OTD (old format) HDF file. The data is returned in a single IDL "structure of structures" variable (see above) *which conforms to the new LIS/OTD data format*. A dialog window monitors the progress of the file input.

Calling Sequence

READ_OLD_ORBIT, *FILENAME*, *ORBIT [, OLD_ORBIT]*

Arguments

FILENAME
The HDF data file *full* file name (i.e., path + name).

ORBIT
A named variable in which the orbit science data will be returned. This need not be declared beforehand.

OLD_ORBIT
An optional named variable in which the orbit science data will be returned in its original ("old OTD") format. Structure definitions for this format can be determined via the `HELP, /STRUCT, OLD_ORBIT` command.

Keywords

NATIVE
Set this keyword to the full path and name of the shared libraries to be used to accelerate file input, if these are available on your platform.

NO_VIEWTIMES
Set this keyword to neglect input of viewtime granules, if shared libraries are not available for your platform, viewtime data is not needed (i.e., lightning *rate* computations are not being performed) and you wish to reduce the file input time.

NO_ONE_SECONDS
Set this keyword to neglect input of one-second data, if shared libraries are not available for your platform, one-second data is not needed (i.e., you do not wish to check alert flags or plot geolocated background images) and you wish to reduce the file input time.

NO_AREAS
NO_FLASHES
NO_GROUPS
NO_EVENTS
Set one or more of these keywords to neglect input of the associated lightning point data, if shared libraries are not available for your platform, the specific point data is not needed and you wish to reduce the file input time.

Examples

Load an old OTD orbit using shared libraries:

```
libs = $
  "/usr/me/LISOTD/IDL/LISOTD_Irix.so"
fn   = $
  "/data/95_194/mlab.otd.1_1.1995.194.0003"
READ_OLD_ORBIT, fn, orbit, NATIVE=libs
```

Load an old OTD orbit using native IDL libraries, neglecting viewtimes to speed things up. Also return the data in old OTD structure format:

```
fn = $
   "/data/95_194/mlab.otd.1_1.1995.194.0003"
READ_OLD_ORBIT, fn, new, old, /NO_VIEWTIMES
```

5.3 Date/Time Utilities

5.3.1 NEW_DATETIME_STRUCTURE

The NEW_DATETIME_STRUCTURE function creates a single instance or an array of instances of the DateTime structure described in (ref) above. These are primarily useful in conjunction with the CALC_DATETIME routine, described below.

Calling Sequence

Result = NEW_DATETIME_STRUCTURE(N)

Arguments

N
The size of the returned DateTime structure array. Setting N=1 returns a single DateTime variable.

Keywords

None

Examples

Create an array of DateTime structures, one for each flash in a loaded orbit file. Load it with the flash TAI93 times:

```
numfl = orbit.point.point_summary.flash_count
dates = NEW_DATETIME_STRUCT(numfl)
dates[*].TAI93 = $
    orbit.point.lightning.event[*].TAI93_time
```

5.3.2 CALC_DATETIME

The CALC_DATETIME procedure takes a partially loaded Date-Time structure variable (or array of structure variables) and attempts to populate the empty fields for other date/time conventions. E.g., if the UTC year, month and day fields are initially set, CALC_DATETIME will calculate the day-of-year field (since no hour/minute/second data are available for the other fields). If the TAI93 field is set, CALC_DATETIME will populate the complete UTC and GPS fields, as TAI93 is a complete date/time variable.

Calling Sequence

CALC_DATETIME,*DATETIME*

Arguments

DATETIME
A date/time structure or array of structures returned from the routine NEW_DATETIME_STRUCTURE, and partially loaded with some date/time information. On return this procedure fills as many empty fields in DATETIME as possible given the input information.

Keywords

LONGITUDE
An optional longitude or array of longitudes from which to compute local times. If DATETIME is passed as an array and LONGITUDE as a scalar, the single longitude will be used for all calculations.

Examples

Compute the day of year from year/month/day:

```
DT = NEW_DATETIME_STRUCTURE(1)
DT.UTC.YEAR  = 1997
DT.UTC.MONTH = 7
DT.UTC.DAY   = 1
CALC_DATETIME, DT
PRINT, DT.UTC.DOY
```

Convert TAI93 times to UTC and local date/times. The times are assumed measured at 84 W, 132 W, and 95 E:

```
DT = NEW_DATETIME_STRUCTURE(3)
DT[*].TAI93 = $
    [47174402.000D, 94608003.000D, 141868804.000D]
CALC_DATETIME, DT, LONGITUDE=[-84.0,-132.0,95.0]
PRINT, DT[*].UTC.YEAR, DT[*].UTC.MONTH, $
       DT[*].UTC.DAY, DT[*].UTC.HOUR, $
       DT[*].UTC.MINUTE, DT[*].UTC.SECOND
PRINT, ' '
PRINT, DT[*].LOCAL.YEAR, DT[*].LOCAL.MONTH, $
       DT[*].LOCAL.DAY, DT[*].LOCAL.HOUR, $
       DT[*].LOCAL.MINUTE, DT[*].LOCAL.SECOND
```

Calculate the UTC and local date/times for all flashes in a LIS/OTD orbit:

```
numfl = orbit.point.point_summary.flash_count
dates = NEW_DATETIME_STRUCT(numfl)
dates[*].TAI93 = $
     orbit.point.lightning.event[*].TAI93_time
lons  = orbit.point.lightning.flash[*].location[1]
CALC_DATETIME, dates, LONGITUDE=lons
```

5.4 Geolocation Utilities

The geolocation utilities described below all assume that some LIS/OTD orbit data is loaded, and stored in a global common block of the form:

```
common datafiles, orbit, fname
```

where `orbit` is the name of the currently loaded orbit variable and `fname` is the orbit's file name (not used by these routines). Note the "One Second" data values must have been included in the loaded orbit for these routines to work.

5.4.1 GET_NADIR_LOCATION

The GET_NADIR_LOCATION and GET_NADIR_LOCATIONS procedures are fast routines to calculate the Microlab-1 or TRMM nadir points (on the Earth's surface) for a selected TAI93 time or array of TAI93 times.

Calling Sequence

GET_NADIR_LOCATION, *TAI93_TIME*, *LOCATION*
GET_NADIR_LOCATIONS, *TAI93_TIMES*, *LOCATIONS*

Arguments

TAI93_TIME
TAI93_TIMES
A scalar TAI93 value or array of TAI93 values specifying the times at which the satellite nadir locations are required.

LOCATION
LOCATIONS
A *2x1* or *2xn* array of computed lat/lon values returned by the routines.

Keywords

None

Examples

Compute and plot satellite nadir locations for each second of an orbit:

```
GET_NADIR_LOCATIONS, $
    orbit.one_second[*].TAI93_time, $
    locations
plot, locations[1,*], locations[0,*]
```

5.4.2 GET_POINTING_VECTOR

The GET_POINTING_VECTOR, GET_VECTOR_POINTING_VECTOR and GET_ARRAY_POINTING_VECTOR procedures are routines to calculate the scaled pointing vector of a pixel, vector of pixels, or the full array of pixels in the OTD/LIS sensor arrays. The scalar version of the routine is meant for single pixel use. The vector version is most useful for a list of pixels. The array version is most useful for geolocating background images. These routines are meant to be used in conjunction with the GET_EARTH_INTERSECTION family of routines, described below.

Calling Sequence

GET_POINTING_VECTOR,*SENSOR*,*X*,*Y*,*LOOK_VECTOR*
GET_VECTOR_POINTING_VECTOR,*SENSOR*,*X*,*Y*,*LOOK_VECTOR*
GET_ARRAY_POINTING_VECTOR,*SENSOR*,*LOOK_VECTOR*

Arguments

SENSOR
A string denoting which sensor the computations are relevant for, e.g., "OTD" or "LIS".

X
Y
The *(x,y)* pixel locations (0 to 127) of interest. These are either a scalar or vector list of values. These arguments are not used in the array version of the routine as the computation is performed over the entire 128x128 array.

LOOK_VECTOR
The calculated pointing vector, returned as a double precision *1x3*, *nx3* or *128x128x3* array. This should be passed directly to one of the GET_EARTH_INTERSECTION family of routines.

Keywords

None

Examples

See below.

5.4.3 GET_EARTH_INTERSECTION

The GET_EARTH_INTERSECTION, GET_VECTOR_EARTH_INTERSECTION and GET_ARRAY_EARTH_INTERSECTION procedures are fast routines to calculate the LIS or OTD ground or cloud-top locations for a selected TAI93 time or array of TAI93 times.

Calling Sequence

GET_EARTH_INTERSECTION,*LOOK_VEC*,*TAI93*,*LL*,*PRLX*,*RET*
GET_VECTOR_EARTH_INTERSECTION,*LOOK_VEC*,*TAI93*,*LL*,*PRLX*,*RET*
GET_ARRAY_EARTH_INTERSECTION,*LOOK_VEC*,*TAI93*,*LL*,*PRLX*,*RET*

Arguments

LOOK_VEC
A pointing vector array returned by one of the GET_POINTING_VECTOR family of routines described above.

TAI93
A single TAI93 time (scalar or array routine) or array of TAI93 times (vector routine) for which to geolocate the requested pixel(s).

LL
A *1x2*, *nx2* or *128x128x2* array of geolocated lat/lon pairs computed by the routine.

PRLX
The height, in meters, above the earth surface to compute the geolocation. Locations precomputed in the HDF files assume that background images are geolocated at the surface and lightning events at cloud top (assumed 12000m). Users may wish to adjust these values as they see fit and regeolocate certain data.

RET
A status variable giving the success or failure of the geolocation calculation; non-zero values indicate failure.

Keywords

None

Examples

Re-geolocate all events in a file assuming 16 km cloud tops:

```
get_vector_pointing_vector, 'lis', $
    orbit.point.lightning.event[*].x_pixel, $
    orbit.point.lightning.event[*].y_pixel, $
    look_vector
get_vector_earth_intersection, look_vector, $
    orbit.point.lightning.event[*].TAI93_time, $
    new_locations, 16000.0D, retval
plots, new_locations[1], new_locations[0], psym=3
```

Geolocate a background image:

```
this_bg = orbit.point.bg_summary[12]
get_array_pointing_vector, 'lis', look_vector
get_array_earth_intersection, look_vector, $
    this_bg.TAI93_time, array_locations, $
    0.0D, retval
; You now have an array of lat lon values -
;   there are many different ways of creating
;   a background image plot from these ...
;   try some!
```

5.5 General Purpose Utilities

5.5.1 WHICH_SENSOR

The WHICH_SENSOR procedure probes an HDF file and determines if it is an old (OTD) or new (LIS/OTD) format file.

Calling Sequence

WHICH_SENSOR,*FILENAME*,*SENSOR*

Arguments

FILENAME
A full filename (path+name) to an OTD or LIS/OTD HDF file.

SESNOR
A variable which will be set to 'lis', 'otd' or 'unknown' upon completion of this routine.

Keywords

None

Examples

Load a mystery HDF file using the correct IDL routine:

```
mysteryfile = "/data/mysteryfile.hdf"
which_sensor, mysteryfile, sensor
case sensor of
   "lis": begin
             read_orbit, mysteryfile, orbit
          end
   "otd": begin
             read_old_orbit, mysteryfile, orbit
          end
   "unknown": begin
             print, "No clue what this file is!"
          end
endcase
```

5.5.2 OTD_QA

The OTD_QA function returns the Quality Assurance flags set by the LIS/SCF QA inspector for OTD data. This is retrieved from a periodically- updated database (`OTD_QA.dat`) distributed with the LIS/OTD software package. The current database runs through 31 December 1997. The flags are returned in a 5-value byte array.

Calling Sequence

retval = OTD_QA(*FILENAME*,*DATABASENAME*)

Arguments

FILENAME
A partial or full (path+name) file name of an OTD HDF file.

DATABASENAME
The full (path+name) file name of the QA database.

Keywords

None

Notes

The five returned values each represent a different possible warning/error condition manually set by the LIS/SCF QA inspector. Assuming the flags are returned in the variable `retval`, then the interpretations are as follows:

- `retval[0] = 1` : This file contained no usable data, and it was removed from the data set distributed to the user community. If you're outside the LIS/SCF, you should never see this flag!
- `retval[1] = 1` : This file contains a large number of events that appear not to have been caused by lightning.
- `retval[2] = 1` : This file contains undocumented data gaps causing view time to be incorrect. Gaps are *confirmed*.

- `retval[2] = 2` : This file may contain undocumented data gaps causing view time to be incorrect. Gaps are *possible*.

- `retval[2] = 3` : This file appears to contain more than one Microlab orbit due to bad production code processing; use lightning data beyond the nominal end of the orbit with caution, as it may not contain corresponding viewtimes.

- `retval[3] = 1` : This file contains data beyond the normal orbit length or the file is tagged with a bad start or stop time.

- `retval[4] = 1` : The ephemeris in this file is questionable and/or the satellite is out of control.

Examples

Check the QA flags for an OTD orbit; if no flags have been set, go ahead and load it:

```
fn = $
    "/data/95_194/mlab.otd.1_1.1995.194.0003"

qaflags = OTD_QA(fn,"/data/OTD_QA.dat")

if (total(qaflags[*]) eq 0) then begin
   READ_OLD_ORBIT, fn, orbit, /QUIET
endif
```

Chapter 6

C Interface: High-level API

The C language high-level LIS/OTD API consists of several main components: a set of C structure variable definitions to hold LIS/OTD data, a simple procedure for LIS/OTD HDF file access and several useful utility routines. Together, they should facilitate rapid development of custom C extraction and analysis code by LIS/OTD end-users. To use the API, familiarity with the basic LIS/OTD data structure and the basics of C programming is required.

6.1 History

When the Microlab-1 satellite was launched in 1995, a C language API for analysis of OTD data (`ReadOTD`) was released to the user community, supported primarily on SGI Irix platforms. This API was designed such that users could write entire analysis programs with just a few high-level library routines included in the package. Over time, its functionality grew beyond mere data extraction to include the repair of some defects in the distributed OTD data files, custom filtering, subsetting and coincidence searches, etc. This greater complexity made it increasingly less likely that the code would be supported on multiple platforms, and degraded the code performance.

From the start, the high-level routines in `ReadOTD` were designed to be flexible enough to accomodate new sensors (such as LIS) without changing the high level interface. Rather, any necessary code changes could be made in the low-level library routines, which users need never access. This is precisely the approach taken now. The only major change is that users must familiarize themselves with the new LIS/OTD data structures, which we believe are a significant improvement over our earlier data organization.

As discussed in Chapter 1, we also are striving to make the API as cross-platform deployable as possible. As such, the initial LIS/OTD software release

includes only the most basic routines found in `ReadOTD`. The routines giving advanced functionality (e.g., filtering, subsetting, etc) will be reintroduced as they are updated to accomodate the new data structure *and* be more cross-platform deployable. Finally, since LIS format HDF files by design will not require the same repair work performed transparently by `ReadOTD`, code performance (especially in file I/O) should be improved dramatically.

6.2 Compatibility

The LIS/OTD high-level C API is completely compatible with old-format (OTD) HDF files and new-format (LIS/OTD) HDF files. The same interface routine `GetData()` is used to access either file type. Old OTD data is converted to the new LIS/OTD data format before being returned to the user.

If you have analysis programs written with the `ReadOTD` package and wish to update them for use with the new API, there are several considerations:

1. You must change your program `#include`s and basic variable declarations to use the new files; a basic program template may be found in Appendix B.

2. Note that some subsetting and filtering routines are not yet functional for LIS data (e.g., the old `SetBounds()` family of routines). However, they are still functional if you use the new libraries to read old OTD format files. Further, programs including these calls will compile even if you wish to access the new LIS format files - it is only the subsetting *functionality* which is not enabled for LIS format files. Thus your old code should be almost completely recompilable using the new libraries.

3. However, since the basic data type returned by `GetData()` has changed, any custom C code you have added (i.e., anything which doesn't utilize the high level routines, such as your own output, summation, gridding, averaging, etc algorithms) will need to be updated to use the new data format (see below). This is basically just a matter of learning where old OTD data elements have been moved in the new format, and globally replacing the appropriate variable names. Appendix C is your guide to this part of the process.

The transition won't be completely seamless, but we hope the added benefits of the new unified and simplified data and file formats outweigh the inconvenience of the upgrade process.

6.3 Data Structures

Structure variables in C are convenient ways of organizing multiple data types and values in single variables. Structures may also be nested; i.e., we may create a "structure of structures". This capability is used to recreate almost exactly the data organization found in LIS/OTD HDF data files, and implemented in the IDL high-level API.

Users need only be familiar with two data structures:

- A `lis_orbit` data structure which contains the science data of an entire HDF file. The science data (typically 1-3 Mb per file) is loaded using the `GetData()` routine (see below). The `lis_orbit` data structure contains several nested structures:

```
struct lis_orbit {
    struct lis_orbit_summary     orbit_summary;
    struct lis_point             point;
    struct lis_one_second        *one_second;
};

struct lis_point {
    struct lis_point_summary     point_summary;
    struct lis_lightning         lightning;
    struct lis_viewtime          *viewtime;
    struct lis_bg_summary        *bg_summary;
};

struct lis_lightning {
    struct lis_area              *area;
    struct lis_flash             *flash;
    struct lis_group             *group;
    struct lis_event             *event;
};
```

and each data element may be accessed directly, e.g.:

```
/* Compute the total radiance of all flashes
    in an orbit                                  */

num_fl = orbit.point.point_summary.flash_count;
total_radiance = 0.0;

for (i=0; i < num_fl; i++) {
    total_radiance +=
```

```
            orbit.point.lightning.flash[i].radiance;
    }

    printf("%f\n", total_radiance);
```

The complete data elements for each substructure (e.g., radiance, location, etc) are summarized in Appendix C.

- The `aUTCDateTime` data structure containing the subfields of a UTC date/time identifier:

```
struct aUTCDateTime {
        int year, doy, month, day, hour, minute;
        double second;
};
```

(Note `doy` is "day of year", useful in sequential ordering applications).

In addition, the structure type `aBoundset` is used to contain filtering and subsetting preferences and will need to appear in your program declarations. However, variables of this type are all modified directly by high-level LIS/OTD API routines, and users need not worry about the actual contents of `aBoundset` variables.

6.4 Startup/Shutdown Routines

These subroutines are necessary components of any code using the LIS/OTD C API, and should be called at the beginning and end of each program.

6.4.1 Initialize()

NAME

Initialize

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void Initialize();

DESCRIPTION

`Initialize` is the required first statement in any user code written with the LIS/OTD C API. It primarily initializes global variables which are normally transparent to the user.

6.4.2 ResetAllBounds()

NAME

ResetAllBounds

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void ResetAllBounds (**struct** aBoundset *Bounds);

DESCRIPTION

The `ResetAllBounds` routine should be called immediately after `Initialize()` to ensure that the subsetting/bounds structure is properly initialized. It may also be called after successive orbit reads if the bounds are to be reset to their initial state (no subsetting). The `*Bounds` variable must be declared in the main block of the code, but its elements are never directly modified by the user.

6.4.3 FreeData()

NAME

FreeData

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void FreeData (**struct** lis_orbit *TheData);

DESCRIPTION

The `FreeData` routine frees all memory allocated by `GetData()` for the array portions of the `*TheData` orbit variable, i.e., viewtimes, background summaries, areas, flashes, groups and events. The variable may then be used to load a new orbit. This may be useful on memory-limited systems if multiple orbits are being read by the same code.

6.4.4 Example of startup/shutdown sequence

```
# include "liblisotd_read_LISOTD.h"

main(int argc, char *argv[]) {

    struct aBoundset Bounds;
    struct lis_orbit TheData;

    Initialize();
    ResetAllBounds(&Bounds);

    /* Main body of code goes here */

    FreeData(&TheData);

}
```

6.5 Interface Routines

These routines are the primary I/O components of the LIS/OTD C API.

6.5.1 GetData()

NAME

GetData

SYNOPSIS

include "liblisotd_read_LISOTD.h"

struct lis_orbit GetData(**char** hdfname[], **struct** aBoundset *Bounds);

DESCRIPTION

The `GetData` routine loads an entire orbit's science data from an OTD or LIS/OTD format HDF file. The HDF format is currently detected by the existence of the character sequences "mlab" (OTD) or "SC" (LIS) within the HDF file name passed to `GetData`. Although not yet implemented, the data will be subsetted and/or filtered (and appropriate parameters recalculated) according to the bounds previously specified by the user. The data is returned in a variable of type `struct lis_orbit` which must previously have been declared by the user.

The `GetData` routine is quite fast for LIS/OTD format files, but can be rather slow for old OTD format files. This is because of some extensive integrity checking and repair work that is performed transparently on old OTD format data. Future releases of this software will strive to improve `GetData` performance with old data.

6.5.2 WriteData()

NAME

WriteData

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void WriteData(**struct** lis_orbit TheData, **int** output_type, **char** filename[132], **int** dump_or, **int** dump_ar, **int** dump_fl, **int** dump_gr, **int** dump_ev);

DESCRIPTION

The `WriteData` routine is a flexible mechanism for exporting LIS or OTD data in a variety of formats. Currently, we allow data to be rewritten in either HDF or several ASCII formats. Future releases will allow export of background images in HDF Scientific Data Set (SDS) format.

The `output_type` variable determines the format of the exported data. Currently available options are MINI_HDF, ASCII_SINGLE and ASCII_TREE. MINI_HDF files are written in the standard LIS / OTD format (these may be useful for keeping track of subsetted or filtered data, when this functionality is included in the package). ASCII_SINGLE files are separate, space-delimited column ASCII data files, one for each point data type (area, flash, group, event; see below) requested. ASCII_TREE files are single files containing the requested point data in a nested "tree" format, retaining the point data grouping hierarchy. The content and format of the column data in these files is controlled by several auxiliary routines, described below.

The `filename` parameter is actually a file name "stub"; appropriate suffixes will be appended to this by `WriteData` to indicate the type and content of the output files created.

The `dump_*` variables are flags denoting whether or not to include the specified point data level in the output (primarily with ASCII format output). Predefined keys are available for use here, e.g., the sequence "`...!O, A, F, !G, !E...` would indicate that only **A**reas and **F**lashes (not, or "!" **O**rbits, **G**roups and **E**vents) are desired for output.

6.5.3 AddASCIIOutputField()

NAME

AddASCIIOutputField

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void AddASCIIOutputField (**int** data_level, **int** output_field, **char** format[8])

Field	A	F	G	E	Type	Units
INDEX	√	√	√	√	long	(none)
THRESH	√	√	√	√	byte	8-bit counts
TIME_UTC	√	√	√	√	misc	Y JD M D H M S.SSS
TIME_TAI93	√	√	√	√	double	seconds
TIME_GPS	√	√	√	√	double	seconds
LOCATION	√	√	√	√	double	degrees
DURATION	√	√			double	seconds
VIEW_TIME	√	†	†	†	double	seconds
RADIANCE	†	√	√	√	double	uJ/ster/m2/ster
FOOTPRINT	†	√	†	†	double	km2
NUM_CHILDREN	√	√	√		long	(none)
NUM_GRANDCHILD	†	†			long	(none)
NUM_GREATGRANDCHILD	†				long	(none)
ALERT_FLAG	†	†	†	†	byte	packed bits
CLUSTER_INDEX	†	†	†	†	byte	(none)
DENSITY_INDEX	†	†	†	†	byte	(none)
NOISE_INDEX	†	†	†	†	byte	(none)
AMPLITUDE				†	byte	7-bit counts
SZA_INDEX				†	byte	-
GLINT_INDEX				†	byte	-

Table 6.1: Data available for ASCII output. √ indicates the value is available in the current version of the API, † indicates the value will be available in the final API release. Not all values are available for all lightning data levels.

DESCRIPTION

This routine adds custom output fields to the files created when `WriteData` is called with options of either ASCII_SINGLE or ASCII_TREE. The `data_level` parameter indicates areas, flashes, groups or events and is specified by passing the keywords `A`, `F`, `G` or `E`. Keywords for the available output fields are shown in the table below. The format string may contain a custom format (C language style) or the key `DEFAULT` to use formats recommended by the LIS-SCF.

6.5.4 ResetASCIIOutputFields()

NAME

ResetASCIIOutputFields

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void ResetASCIIOutputField (**int** data_level, **char** reset_value[8])

DESCRIPTION

By default, the ASCII output options of `WriteData()` export the UTC date/time, latitude and longitude of a given data level (**A**rea, **F**lash,**G**roup or **E**vent). Additional fields may be added with the `AddASCIIOutputField()` routine. The `ResetASCIIOutputFields()` routine restores the exported fields to their initial state. The `data_level` parameter is one of the `A`, `F`, `G` or `E` keys. The `reset_value` parameter may be one of two predefined keywords, `DEFAULT` (UTC/lat/lon fields) or `EMPTY` (no fields).

6.6 Subsetting Routines

As discussed above, subsetting and/or filtering functionality is not yet supported in the LIS/OTD software package. Data subsetting/filtering technically requires the recomputation of several value-added parameters, such as radiance and footprint. We are currently investigating efficient ways of including this functionality without impacting code performance for users who do not choose to filter or subset.

You are of course free to write your own subsetting and filtering routines, however we caution that not all lightning value-added parameters at each data level will *necessarily* remain correct, including the viewtime granules. Subsetting and filtering routines will be re-introduced into the LIS/OTD software package as time and resources at the LIS/SCF allow.

6.7 Date/Time Utilities

These are a few useful routines we have included to convert between the UTC, TAI93 and GPS date/time formats.

6.7.1 UTC_to_TAI93()

NAME

UTC_to_TAI93

SYNOPSIS

include "liblisotd_read_LISOTD.h"

double UTC_to_TAI93 (**struct aUTCDateTime** UTC)

DESCRIPTION

This routine converts a *completely specified* UTC date/time to its TAI93 time equivalent. Leap seconds are considered; the last leap second known to these routines was on 1 July 1997.

6.7.2 TAI93_to_UTC()

NAME

TAI93_to_UTC

SYNOPSIS

include "liblisotd_read_LISOTD.h"

struct aUTCDateTime TAI93_to_UTC (**double** TAI93)

DESCRIPTION

This routine converts a TAI93 time (such as those associated with the LIS/OTD point data) to a *completely specified* UTC date/time. Leap seconds are considered; the last leap second known to these routines was on 1 July 1997.

6.7.3 UTC_to_GPS()

NAME

UTC_to_GPS

SYNOPSIS

include "liblisotd_read_LISOTD.h"

double UTC_to_GPS (**struct aUTCDateTime** UTC)

DESCRIPTION

This routine converts a *completely specified* UTC date/time to its GPS time equivalent. Leap seconds are considered; the last leap second known to these routines was on 1 July 1997. Note that GPS time is specified only as a number of seconds since a reference date/time. There is *no such thing* as a GPS time in year/month/day format. Dates/times in such a format exported by GPS receivers have *already* been converted to UTC. Raw "GPS seconds" are rarely used outside of the satellite community.

6.7.4 GPS_to_UTC()

NAME

GPS_to_UTC

SYNOPSIS

include "liblisotd_read_LISOTD.h"

struct aUTCDateTime GPS_to_UTC (**double** GPS)

DESCRIPTION

This routine converts a GPS time to its *completely specified* UTC date/time equivalent. Leap seconds are considered; the last leap second known to these routines was on 1 July 1997. Note that GPS time is specified only as a number of seconds since a reference date/time. There is *no such thing* as a GPS time in year/month/day format. Dates/times in such a format exported by GPS receivers have *already* been converted to UTC. Raw "GPS seconds" are rarely used outside of the satellite community.

6.7.5 getDayOfYear

NAME

getDayOfYear

SYNOPSIS

include "liblisotd_read_LISOTD.h"

int getDayOfYear (**int** day, **int** month, **int** year)

DESCRIPTION

This routine converts a day/month/year date specification to a 3-digit day of year, as is contained in the LIS/OTD HDF file names.

6.7.6 InvJulian

NAME

InvJulian

SYNOPSIS

include "liblisotd_read_LISOTD.h"

void InvJulian (**int** year, **int** day_of_year, **int** *month, **int** *day)

DESCRIPTION

This routine converts a year/day of year date specification to a month and a day.

6.8 Geolocation Utilities

Geolocation utilities (primarily used for geolocating background images) are not included in this prerelease but will be implemented in the official release of the LIS/OTD software. Until then, adventurous programmers should be able to easily port the geolocation routines included in the IDL API (see Section 5.4). Note that unlike the earlier `ReadOTD` software, the LIS/OTD software utilizes no proprietary or third-party libraries to perform geolocation functions.

Part III

Low Level Interfaces

Chapter 7

C Interface: Low-level API

We only provide limited documentation of the LIS portion of the low level C API in this prerelease version of the documentation. Low-level API programs can be compiled using the `make reader_lowlevel` option of the `Makefile` we provide. The low level API is still fairly straightforward; the main differences being:

- Certain date/time and geolocation utility functions are not accessible
- The user must handle HDF file opening and closing
- The user must handle memory allocation and deallocation for structure variable arrays
- The returned HDF data is not cleanly assembled into a single structure variable, but rather a number of individual structure variable arrays
- There will be no eventual support for filtering and subsetting

The primary benefit of the low-level API is that it should be extremely portable across platforms, as it uses fairly vanilla C code and the NCSA HDF libraries. If all vdatas in an HDF file are to be imported, there is no significant speed benefit to using the low-level API.

The following sample program illustrates the basic structure and interfaces of the low-level API. The individual vdata structure definitions are the same as in the high-level API, as described in Chapter 6 and Appendix C.

```
/* Basic includes you should have in any program
   using the low-level API */

# include <stdio.h>
# include <math.h>
# include "hdf.h"
# include "liblisotd_err_defs.h"
# include "liblisotd_readhdf.h"

main(int argc, char *argv[]) {

    long fid,i;
    long lis_events, lis_one_seconds, lis_areas;
    struct lis_area          *lis_area_var;
    struct lis_event         *lis_event_var;
    struct lis_one_second    *lis_one_second_var;
    char fname[132];

    /* This is the file we'll access; it lives in the
       same directory as the executable. */

    sprintf(fname,"TRMM_LIS_SC.03.4_1998.028.00970");

    /* Get the file ID of the LIS HDF data file.
       Some orbits may not have metadata inserted
         into them, or may legitimately have no
         point vdatas.  Errors are encoded in the
         file ID's if this occurs.  We mask them
          out in order to continue. */

    fid = OPEN_lis_orbit_hdf(fname);

    if (((fid&0xffffff00)==lis_NoMetadataMissingVdata)) {
                fid = fid^lis_NoMetadataMissingVdata;
    } else if (((fid&0xffffff00)==lis_NoMetadata)) {
                fid = fid^lis_NoMetadata;
    } else if (((fid&0xffffff00)==lis_MissingVdata)) {
                fid = fid^lis_MissingVdata;
    }

    /* Should have a valid file ID at this point.
        If not, we're in trouble. */

    if (fid < 0) {
            printf("Unrecoverable error opening %s\n",fname);
            if(CLOSE_lis_orbit_hdf(fid)!=lis_operation_SUCCESSFUL)
```

```
                printf("close %s file failed\n",fname);
            exit(-1);
    }

    /* Get the number of area vdata records */

    lis_areas = lis_area_COUNT(fid);

    /* Allocate memory accordingly */

    lis_area_var = (struct lis_area *)
        malloc((size_t)(lis_areas*sizeof(struct lis_area)));

    /* Read the area vdata i from the file.  It will be
        contained in the variable lis_area_var, which
        follows the conventional structure definition.
        See Appendix C of the documentation or the file
        liblisotd_readhdf.h for details. */

    i = READ_lis_areas(fid,lis_area_var,0,lis_areas);

    /* Exactly the same for events */

    lis_events = lis_event_COUNT(fid);
    lis_event_var = (struct lis_event *)
        malloc((size_t)(lis_events*sizeof(struct lis_event)));
    i = READ_lis_events(fid,lis_event_var,0,lis_events);

    /* And for one second data, etc. */

    lis_one_seconds = lis_one_second_COUNT(fid);
    lis_one_second_var = (struct lis_one_second *)
        malloc((size_t)(lis_one_seconds*sizeof(struct lis_one_second)));
    i = READ_lis_one_seconds(fid,lis_one_second_var,0,lis_one_seconds);

    /* You get the idea.  Now close out the orbit nicely and
        free the allocated memory. */

    if(CLOSE_lis_orbit_hdf(fid)!=lis_operation_SUCCESSFUL)
        printf("close %s file failed\n",fname);

    free(lis_area_var);
    free(lis_event_var);
    free(lis_one_second_var);
}
```

Part IV

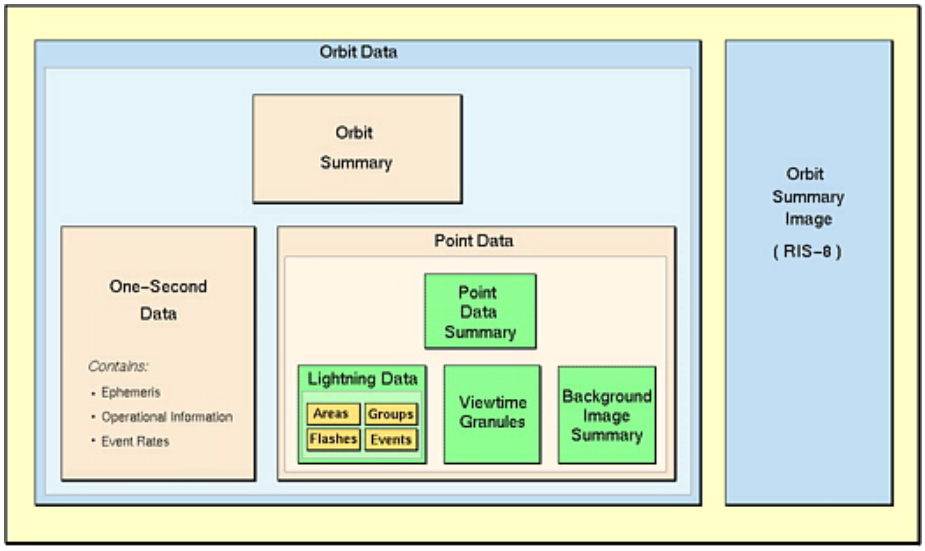
Appendices

Appendix A

HDF/C/IDL Structures

The basic LIS/OTD data file structure has already been partially documented in Chapters 5 and 6. This Appendix provides detailed information on the individual fields of each structure. In all cases, the field names are identical in the HDF, C and IDL implementations of the data structure. The overall organization of the data is shown below, and the fields desribed in the sections that follow.

A.1 Basic structure



A.2 orbit_summary

Parameter	Structure Element Name	Type	Size	Description	Units
Orbit ID	id_number	int32	1	the number of this orbit, where the orbit count starts at launch	-
TAI93 start time	TAI93_start	float64	1	start of this orbit in TAI93 time	sec
UTC start time	UTC_start	char	28	start of this orbit in UTC time	-
GPS start time	GPS_start	float64	1	start of this orbit in GPS time	sec
TAI93 end time	TAI93_end	float64	1	end of this orbit in TAI93 time	sec
start longitude	start_longitude	float32	1	longitude boundary defining start of this orbit	deg
end longitude	end_longitude	float32	1	longitude boundary defining end of this orbit	deg
number of point data records	point_data_count	int16	1	number of point data elements associated with this orbit	-
point data child record number	point_data_address	int16	1	address of the first element in point data structure	-
number of one second records	one_second_count	int32	1	number of elements in the one-second data	-
one-second child record number	one_second_address	int32	1	address of the first element in the one-second data	-
number of summary GIF images	summary_image_count	int16	1	number of summary GIF images	-
summary GIF image record number	summary_image_address	int16	1	address of the first summary GIF image	-
inspection code	inspection_code	int16	1	code indicating which problem scenarios were checked by the QA inspector	-
configuration code	config_code	int16	1	code indicating which code configuration scenario was used when processing the data	-

Notes

- **Inspection Code**: This field is used to identify which QA procedure was followed at the time of the QA inspection. It is expected that the QA procedure will evolve as more data is collected and new "problems" are

identified. As a result, each orbit must be marked so that the user (and QA inspector) can identify which orbits were checked for specific "problems".

- **Configuration Code**: This field is used to identify when a change is made to the code that processed the data, such as changes in the input parameters to the processing code. Each orbit must be marked so that the user (and QA inpector) can identify which orbits were processed using specific input parameters.

- **Start, End Longitude**: The start longitude is defined as the satellite nadir longitude (rounded to the nearest 0.5 degrees) at the time when the satellite is at its southernmost point in the orbit. The end longitude is calculated in the same manner, and it is the same longitude as the start longitude for the following orbit.

- **TAI93 Start, End Time**: The start time corresponds with the time which the leading edge of the instrument field-of-view crosses the start longitude, which is BEFORE the time of the satellite nadir crossing of this longitude. Similarly, the end time corresponds with the time that the trailing edge of the instrument field-of-view crosses the end longitude, which is AFTER the time of the satellite nadir crossing of this longitude. Note that there will be some overlap in the time spans of successive orbits.

A.3 one_second

Parameter	Structure Element Name	Type	Size	Description	Units
TAI93 time	TAI93_time	float64	1	whole second value starting before and continuing beyond one orbit	sec
Alert summary flag	alert_summary	char	1	bit masked summary of alert flags (see below)	-
Instrument alert flag	instrument_alert	char	1	bit masked status of instrument (see below)	-
Platform alert flag	platform_alert	char	1	bit masked status of platform (see below)	-
External alert flag	external_alert	char	1	bit masked status of external factors (see below)	-
Processing alert flag	processing_alert	char	1	bit masked status of processing algorithms (see below)	-
Platform coordinates	position_vector	float32	3	location of platform in ECR coordinates	m
Platform velocity	velocity_vecotr	float32	3	velocity of platform in ECR coordinates	m
Transform matrix	transform_matrix	float32	9	components of transform from pixel plane-boresight coordinates to ECR coordinates of boresight and pixel plane	-
Solar vector	solar_vector	float32	3	unit vector from center of earth to sun in ECR coordinates	
Ephemeris quality flags	ephemeris_quality_flag	int32	1	Bit masked status (see below)	-
Attitude quality flags	attitude_quality_flag	int32	1	Bit masked status (see below)	-
Threshold estimate	boresight_threshold	char	1	Most likely threshold value applied to the boresight position given the solar zenith angle, assuming clouds present	-
8-bit threshold values	thresholds	char	16	values of the instrument threshold settings for each 256 count background interval	-
Noise index (percent signal)	noise_index	char	1	a metric that indicates the noise level	%*100
Event count	event_count	int16	6	raw event count and counts surviving filters at each processing stage	-

A.4 point_summary

Parameter	Structure Element Name	Type	Size	Description	Units
Parent record number	parent_address	int32	1	address of parent (orbit)	-
Number of events	event_count	int32	1	number of events in orbit	-
Event record number	event_address	int32	1	address of first event	-
Number of groups	group_count	int32	1	number of groups in orbit	-
Group record number	group_address	int32	1	address of first group	-
Number of flashes	flash_count	int32	1	number of flashes in orbit	-
Flash record number	flash_address	int32	1	address of first flash	-
Number of areas	area_count	int32	1	number of areas in orbit	-
Area record number	area_address	int32	1	address of first area	-
Number of backgrounds	bg_count	int32	1	number of background image summary records in orbit	-
Backgorund image summary record number	bg_address	int32	1	address of first background image summary	-
Number of viewtime granules	vt_count	int32	1	number of viewtime granules in orbit	-
Viewtime granule record number	vt_count	int32	1	address of first viewtime granule in orbit	-

A.5 viewtime

Parameter	Structure Element Name	Type	Size	Description	Units
Grid cell position	location	float32	2	lat/lon of the center of the grid cell of dimensions 0.5 deg x 0.5 deg	deg
Start time	TAI93_start	int32	1	TAI93 whole second when loaction was first within FOV	sec
End time	TAI93_end	int32	1	TAI93 whole second when location was last within FOV	sec
Effective viewtime	effective_obs	float32	1	time of observation of the grid cell, adjusted by the percentage of area in the grid cell within the FOV	sec
Alert flag	alert_flag	char	1	reflects status of instrument, platform, external factors and processing algorithms	-
8-bit threshold	approx_thresohld	char	1	threshold of instrument corresponding with grid cell position, proxied from solar zenith angle at a time halfway between start and end time	-

A.6 bg_summary

Parameter	Structure Element Name	Type	Size	Description	Units
TAI93 time	TAI93_time	float64	1	TAI93 time of the background image	sec
Background image number	address	int32	1	image number within orbit	-
Boresight position	boresight	float32	2	lat/lon location of center pixel (63,64)	deg
Corner positions	corners	float32	8	lat/lon locations of corner pixels	-

A.7 area

Parameter	Structure Element Name	Type	Size	Description	Units
Area time	TAI93_time	float64	1	TAI93 time of 1st event in area	sec
Point data time span	delta_time	float32	1	time between first and last event that compose the area	sec
Duration of observation	observe_time	int16	1	duration of observation of the region where the area occurred	sec
Geolocated position	location	float32	2	lat/lon radiance-weighted centroid	deg
Calibrated radiance	net_radiance	float32	1	sum of event radiances composing this area	uJ/ster /m2/um
Footprint size	footprint	float32	1	unique areal extent	sq km
Area record number	address	int32	1	area address	-
Parent record number	parent_address	int32	1	pointer to parent's address (point data)	-
Child record number	child_address	int32	1	address of 1st flash in a sequential list	-
# of children	child_count	int32	1	# of flashes in area	-
# of grandchildren	grandchild_count	int32	1	# of groups in area	-
# of great-grandchildren	greatgrandchild_count	int32	1	# of events in area	-
8-bit threshold	approx_threshold	char	1	estimated value of 8-bit threshold for the area determined from background level or solar zenith angle	-
Alert flag	alert_flag	char	1	bit masked status of instrument, platform, external factors and processing algorithms	-
Clustering probabilit	cluster_index	char	1	pixel density metric; higher numbers indicate area is less likely to be noise	(0-99)
Lightning activity	density_index	char	1	spatial density metric; higher if area geolocated in a region of high lightning activity	-
Noise index	noise_index	char	1	signal-to-signal plus noise ratio	%x100
Eccentricity	oblong_index	float32	1	metric indicating how oblong the area is	-
Time order	grouping_sequence	int32	1	time sequence of area used when grouping algorithm is applied	-
Grouping status	grouping_status	char	1	0=area grouped normally; 1=area split between orbits; 2=area split between orbits; 3=grouping algorithm failed	-

A.8 flash

Parameter	Structure Element Name	Type	Size	Description	Units
Flash time	TAI93_time	float64	1	TAI93 time of 1st event in flash	sec
Point data time span	delta_time	float32	1	time between first and last group that compose the flash	sec
Duration of observation	observe_time	int16	1	duration of observation of the region where the flash occurred	sec
Geolocated position	location	float32	2	lat/lon radiance-weighted centroid	deg
Calibrated radiance	net_radiance	float32	1	sum of event radiances composing this flash	uJ/ster /m2/um
Footprint size	footprint	float32	1	unique areal extent	sq km
Flash record number	address	int32	1	flash address	-
Parent record number	parent_address	int32	1	pointer to parent's address (area)	-
Child record number	child_address	int32	1	address of 1st group in a sequential list	-
# of children	child_count	int32	1	# of groups in flash	-
# of grandchildren	grandchild_count	int32	1	# of events in flash	-
8-bit threshold	approx_threshold	char	1	estimated value of 8-bit threshold for the flash determined from background level or solar zenith angle	-
Alert flag	alert_flag	char	1	bit masked status of instrument, platform, external factors and processing algorithms	-
Clustering probabilit	cluster_index	char	1	pixel density metric; higher numbers indicate flash is less likely to be noise	(0-99)
Lightning activity	density_index	char	1	spatial density metric; higher if flash geolocated in a region of high lightning activity	-
Noise index	noise_index	char	1	signal-to-signal plus noise ratio	%x100
Solar glint cosine	glint_index	float32	1	cosine of angle	-
Eccentricity	oblong_index	float32	1	metric indicating how oblong the flash is	-
Time order	grouping_sequence	int32	1	time sequence of flash used when grouping algorithm is applied	-
Grouping status	grouping_status	char	1	0=flash grouped normally; 1=flash split between orbits; 2=flash split between orbits; 3=grouping algorithm failed	-

A.9 group

Parameter	Structure Element Name	Type	Size	Description	Units
Group time	TAI93_time	float64	1	TAI93 time of 1st event in group	sec
Duration of observation	observe_time	int16	1	duration of observation of the region where the group occurred	sec
Geolocated position	location	float32	2	lat/lon radiance-weighted centroid	deg
Calibrated radiance	net_radiance	float32	1	sum of event radiances composing this group	uJ/ster /m2/um
Footprint size	footprint	float32	1	unique areal extent	sq km
Group record number	address	int32	1	flash address	-
Parent record number	parent_address	int32	1	pointer to parent's address (flash)	-
Child record number	child_address	int32	1	address of 1st event in a sequential list	-
# of children	child_count	int32	1	# of events in group	-
8-bit threshold	approx_threshold	char	1	estimated value of 8-bit threshold for the group determined from background level or solar zenith angle	-
Alert flag	alert_flag	char	1	bit masked status of instrument, platform, external factors and processing algorithms	-
Clustering probabilit	cluster_index	char	1	pixel density metric; higher numbers indicate group is less likely to be noise	(0-99)
Lightning activity	density_index	char	1	spatial density metric; higher if group geolocated in a region of high lightning activity	-
Noise index	noise_index	char	1	signal-to-signal plus noise ratio	%x100
Solar glint cosine	glint_index	float32	1	cosine of angle	-
Eccentricity	oblong_index	float32	1	metric indicating how oblong the group is	-
Time order	grouping_sequence	int32	1	time sequence of group used when grouping algorithm is applied	-
Grouping status	grouping_status	char	1	0=group grouped normally; 1=group split between orbits; 2=group split between orbits; 3=grouping algorithm failed	-

A.10 event

Parameter	Structure Element Name	Type	Size	Description	Units
Event time	TAI93_time	float64	1	TAI93 time of event	sec
Duration of observation	observe_time	int16	1	duration of observation of the region where the event occurred	sec
Geolocated position	location	float32	2	lat/lon radiance-weighted centroid	deg
Calibrated radiance	net_radiance	float32	1	radiance of this event	uJ/ster /m2/um
Footprint size	footprint	float32	1	unique areal extent	sq km
Event record number	address	int32	1	flash address	-
Parent record number	parent_address	int32	1	pointer to parent's address (group)	-
X pixel	x_pixel	char	1	CCD pixel column	-
Y pixel	y_pixel	char	1	CCD pixel row	-
Background illumination	bg_value	int16	1	level of background illumination (16-bit) at time of event	-
Background radiance	bg_radiance	int16	1	background radiance associated with pixel at time of event	W/ster /m2/um
7-bit amplitude	amplitude	byte	1	uncalibrated optical amplitude reported by instrument	-
Solar zenith angle	sza_index	byte	1	solar zenith angle	deg
Solar glint angle	glint_index	byte	1	angle between line of sight vector and direct solar reflection vector	deg
8-bit threshold	approx_threshold	char	1	estimated value of 8-bit threshold for the event; from bg level or solar zenith angle	-
Alert flag	alert_flag	char	1	bit masked status of instrument, platform, external factors and processing algorithms	-
Clustering probabilit	cluster_index	char	1	pixel density metric; higher numbers indicate event less likely to be noise	(0-99)
Lightning activity	density_index	char	1	spatial density metric; higher if event geolocated in a region of high lightning activity	-
Noise index	noise_index	char	1	signal-to-signal plus noise ratio	%x100
Background illumination flag	bg_value_flag	char	1	bg radiance has been 0: estimated from s.z.a. 1: interpolated from bgs	-
Time order	grouping_sequence	int32	1	time sequence of event used when grouping algorithm is applied	-

A.11 Alert Flags

Inside the LIS HDF files, there is a vdata called the **one second** data. The purpose of this data set is to provide temporal information about the LIS instrument and its data on a one-second basis for each second of the TRMM mission. Within the one-second data structure, there are 5 bytes that are set aside for 4 alert flags and 1 summary flag. Each of these flags provides binary information about the quality or accuracy of the data.

The four alert flags are the **Instrument Alert Flag**, the **Platform Alert Flag**, the **External Alert Flag**, and the **Processing and Algorithm Alert Flag**. When any of these one-second flags are non-zero, it indicates that the LIS data quality may be compromised in some fashion during that one-second period. Since there are 8 bits per byte, each flag is used to identify up to 8 different types of problems or situations. Each bit of the alert flags is associated with a specific problem that could affect the quality of the data -OR- is associated with a situation that may help in analyzing the LIS data. The definition/meaning of each bit of each of these alert flags is summarized in the tables below.

Each bit of the alert flags can be categorized as *Fatal*, *Warning*, or *Indifferent*. When a bit that is categorized as fatal is set "high" (the bit is set to 1), no data could have been obtained and/or recorded during the one-second period because of a problem that is known to cause data loss. When a bit that is categorized as a warning bit is set "high", it indicates that a problem occurred, and some (but usually not all) of the data during that one-second interval may have been lost. A bit categorized as indifferent is set "high" when something important occurred during the one-second interval, but its effect on the LIS data is minimal or not known.

In addition to the alert flags, there is also an "Alert Summary Flag". The 8 bits in this flag are used to summarize the information in the four alert flags described above. This flag is propagated to the **viewtime**, **area**, **flash**, **group** and **event** records for easy reference. The TAI93 times of these point data may be used to index the **one second** data to investigate the more detailed alert flag settings.

The definition/meaning of each bit of this summary flag is provided in the next table.

Bit	**Value**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**Alert Level**	**Meaning**
1	1	.	.	.	.	.	.	.	X	*Fatal*	Instrument fatal flag
2	2	.	.	.	.	.	.	X	.	*Warning*	Instrument warning flag
3	4	.	.	.	.	.	X	.	.	*Fatal*	Platform fatal flag
4	8	.	.	.	.	X	.	.	.	*Warning*	Platform warning flag
5	16	.	.	.	X	.	.	.	.	*Fatal*	External fatal flag
6	32	.	.	X	.	.	.	.	.	*Warning*	External warning flag
7	64	.	X	.	.	.	.	.	.	*Fatal*	Processing fatal flag
8	128	X	.	.	.	.	.	.	.	*Warning*	Processing warning flag

A.11.1 Instrument Alert

Bit settings

Bit	Value	8	7	6	5	4	3	2	1	Alert Level	Meaning
1	1	.	.	.	.	.	.	.	X	*Fatal*	Instrument Off
2	2	.	.	.	.	.	.	X	.	*Indifferent*	Instrument Command Executed
3	4	.	.	.	.	.	X	.	.	*Fatal / Warning*	FIFO Buffer Overflow
4	8	.	.	.	.	X	.	.	.	*Warning*	Thresholds set very high
5	16	.	.	.	X	.	.	.	.	*Fatal*	Instrument warming up
6	32	.	.	X	.	.	.	.	.	*Warning*	Improper operating temperatures
7	64	.	X	.	.	.	.	.	.	*Fatal*	Packet gap
8	128	X	.	.	.	.	.	.	.	*Warning*	Data handling problem

Interpreting the settings

- **Instrument off**: When there is a large time gap (greater than 15 seconds) between subequent intstrument data packets, then the instrument is assumed to be "off" during the gap (the period of "no data" between the two packets).

- **Instrument command executed**: This flag is set when the packet header information shows that a new command has been executed. Note that there are several commands that could be sent to the instrument, most of which have little or no effect on the event data.

- **FIFO buffer overflow**: This flag is set following the period when the FIFO overflow bit has been set in the header portion of a packet. Following a FIFO buffer overflow, no data is recorded until the data contained in the FIFO buffer is processed. The FIFO Buffer Overflow bit is set from the time of the event prior to the overflow to the time of the event following the data gap caused by the overflow (or 30 seconds, whichever is shorter). This problem is considered fatal during the data gap (since no event data can be recorded during this period), and just a warning at the second before and the second after the data gap (since some events are recorded during at least a portion of the one second period) data .

- **Thresholds set very high**: The LIS instrument has 16 threshold values, where each threshold corresponds to a different background brightness level. When 10 or more of these 16 thresholds are set at or above 63, then this flag is set "high".

- **Instrument warming up**: This flag is set by the QA inspector since it is too difficult for a software algorithm to detect this condition. When the instrument is turned "on" after being "off" for a long period of time, it may not operate nominally until its temperature exceeds a certain level.

- **Improper operating temperatures**: The temperatures of the primary filter, secondary filter, the sensor head, the optics filter, the focal plane, the controller board, and the power converter are reported by the instrument. If any of these temperatures go above or below the "red" levels, this flag is set.

- **Packet gap**: Occasionally, a problem can occur which results in instrument data packets not being recorded or processed properly. Since instrument data packets should be reported by the instrument every 1 to 2 seconds, this flag is set when there is a time gap greater than 3 seconds between packets. If the time gap is longer than 15 seconds, then the instrument is assumed to be "off" (see above) and this flag is not set.

- **Data handling problem**: When the instrument reports a "data read error" or a "fifo overflow" error, this flag is set only at the time of the instrument packet which reported the error.

A.11.2 Platform alert

Bit settings

Bit	Value	8	7	6	5	4	3	2	1	Alert Level	Meaning
1	1	.	.	.	.	.	.	.	X	*Warning*	No attitude or ephemeris quality flags available
2	2	.	.	.	.	.	.	X	.	*Fatal*	Ephemeris not available
3	4	.	.	.	.	.	X	.	.	*Warning*	Ephemeris possibly inaccurate
4	8	.	.	.	.	X	.	.	.	*Fatal*	Attitude not available
5	16	.	.	.	X	.	.	.	.	*Warning*	Attitude possibly inaccurate
6	32	.	.	X	.	.	.	.	.	*Fatal*	Clock not available
7	64	.	X	.	.	.	.	.	.	*Warning*	Clock possibly inaccurate
8	128	X	.	.	.	.	.	.	.	-	(reserved)

Interpreting the settings

- **No attitude or ephemeris quality flags available**: The TRMM data is provided with a set of binary information that qualifies the accuracy of the attitude and ephemeris associated with the satellite. If this binary information is not available, then this flag is set.

- **Ephemeris not available**: When no ephemeris information is available, then this flag is set "high".

- **Attitude not available**: When no attitude information is available, then this flag is set "high".

- **Attitude possibly inaccurate**: When the ephemeris data has a lowered accuracy (as reported in the TRMM Quality Flags), then this flag is set "high".

- **Clock not available**: This flag is set by the QA inspector when deemed appropriate.

- **Clock possibly inaccurate**: This flag is set when two or more events have the same time stamp and are reported to have occurred in the same pixel. When this happens, it is usually caused by the instrument failing to properly time tag the events.

A.11.3 External alert

Bit settings

Bit	Value	8	7	6	5	4	3	2	1	Alert Level	Meaning
1	1	.	.	.	.	.	.	.	X	*Warning*	Satellite within SAA - Model 1
2	2	.	.	.	.	.	.	X	.	*Warning*	Satellite within SAA - Model 2
3	4	.	.	.	.	.	X	.	.	*Warning*	Direct solar reflection possible within FOV
4	8	.	.	.	.	X	.	.	.	*Indifferent*	TRMM Microwave Imager on
5	16	.	.	.	X	.	.	.	.	*Indifferent*	Precipitation Radar on
6	32	.	.	X	.	.	.	.	.	*Indifferent*	Visible Infrared Scanner on
7	64	.	X	.	.	.	.	.	.	*Indifferent*	Clouds and Earth's Radiant Energy System sensor on
8	128	X	.	.	.	.	.	.	.	-	(reserved)

Interpreting the settings

- **Satellite within SAA - Model 1**: When the nadir location of the satellite is within the boundaries of the SAA (as defined by an empirical model), this flag is set "high". The SAA is a location above the earth where there are high radiation levels are present. Since the LIS instrument is sensitive to radiation, high false event rates are obtained within the SAA.

- **Satellite within SAA - Model 2**: When the nadir location of the satellite is within the boundaries of the SAA (as defined by another empirical model), this flag is set "high".

- **Direct Solar Reflection Possible Within Field-of-view**: This flag is set when the solar zenith angle is below 7 degrees. The value of this sza threshold angle is computed using geometry based on the altitude of the spacecraft (350 km) and the maximum observation angle from boresight (50 deg at corner).

- **TRMM Microwave Imager (TMI) "on"**: This flag is set when the TRMM header information reports that TMI is powered up.

- **Precipitation Radar (PR) "on"**: This flag is set when the TRMM header information reports that the PR is powered up.

- **Visible Infrared Scanner (VIRS) "on"**: This flag is set when the TRMM header information reports that the VIRS is powered up.

- **Clouds and Earth's Radiant Energy System Sensor (CERES) "on"**: This flag is set when TRMM header information reports that the CERES sensor is powered up.

A.11.4 Processing and algorithm alert

Bit settings

Bit	Value	8	7	6	5	4	3	2	1	Alert Level	Meaning
1	1	.	.	.	.	.	.	.	X	*Warning*	QA inspector's warning flag
2	2	.	.	.	.	.	.	X	.	*Fatal*	QA inspector's fatal flag
3	4	.	.	.	.	.	X	.	.	*Fatal*	Data too garbled for software to read
4	8	.	.	.	.	X	.	.	.	*Fatal*	Data set too large to process
5	16	.	.	.	X	.	.	.	.	*Fatal / Warning*	Unforseen software error caused improper reporting of data
6	32	.	.	X	.	.	.	.	.	*Warning*	Grouping algorithm buffer limitation problem
7	64	.	X	.	.	.	.	.	.	*Warning*	Viewtime algorithm failure to accurately determine FOV
8	128	X	.	.	.	.	.	.	.	-	(reserved)

Interpreting the settings

- **QA Inspector's Warning Flag**: This flag is set by the QA Inspector when he/she feels the quality of the data may have been compromised and no other alert flags were set "high" to indicate the reason for the lowered expectation of data quality.

- **QA Inspector's Fatal Flag**: This flag is set by the QA Inspector when he/she feels the quality of the data is very poor, and no other alert flags were set "high" to indicate the reason for the bad data.

- **Data too Garbled for Sofware to Read**: This flag is set by the QA inspector when the production software cannot complete the processing

because the data has been corrupted by something external to the instrument.

- **Data Set Too Large to Process**: This flag is set by the QA inspector when the production software cannot complete the processing because the data is too full of noise, and processing the "nominal" data that follows the noisy data is too difficult or not possible.

- **Unforseen Software Error Caused an Improper Reporting of the Data**: This flag is set by one of the production software programs when something prevents the proper calculation or reporting of any part of the data.

- **Grouping Algorithm Buffer Limitation Problem**: When there are too many events per group, too many groups per flash, or too many flashes per area, for the grouping algorithm to handle, this flag flag is set "high".

- **Viewtime Alogrithm Failure to Accurately Determine Field-of-View**: When the viewtime algorithm has problems determining the correct field-of-view (usually due to the satellite pointing several degrees off nadir), then this flag is set.

Appendix B

Sample Code

This chapter provides some simple, complete programs illustrating the usage of the LIS/OTD software. For each example, samples of IDL and C code are provided which perform the same tasks. Note especially the appropriate C or IDL includes and declarations contained in these sample programs.

B.1 Read an orbit

Using the IDL API

```
; Sample IDL code to read an orbit and
; do nothing with it.

@lisotd.pro
@lisapp_DateTime.pro

PRO sample1

    READ_ORBIT,"TRMM_LIS_SC.03.4_1998.026.00933", $
        orbit, /QUIET

END
.
```

Using the C API

```
/* Sample C code to read an orbit and
   do nothing with it.*/

#include <stdio.h>
#include <math.h>
#include "liblisotd_read_LISOTD.h"

main(int argc, char *argv[])
{
    struct lis_orbit TheData;
    struct anOTDBoundset Bounds;

    Initialize();
    ResetAllBounds(&Bounds);

    TheData = GetData("TRMM_LIS_SC.03.4_1998.026.00933",
                      &Bounds);

    FreeData(&TheData);

}
```

B.2 Export some data to ASCII

Using the IDL API

```
; Sample IDL code to read an orbit and
; export the flashes' UTC date/time,
; latitude, longitude and radiance to
; an ASCII file.

@lisotd.pro
@lisapp_DateTime.pro

PRO sample2

    READ_ORBIT,"TRMM_LIS_SC.03.4_1998.026.00933", $
        orbit, /QUIET, /NO_AREAS, /NO_GROUPS, $
        /NO_EVENTS, /NO_VIEWTIMES, /NO_ONE_SECONDS

    ; Define the ouput format, MM/DD/YYYY HH:MM:SS.SSS LL LL R

    flash_format = '((I2.2),"/",(I2.2),"/",(I4)," ",' + $
                   '(I2.2),":",(I2.2),":",(F6.3)," ",' + $
                   '(F7.2)," ",(F7.2)," ",(F8.0))'

    ; Always use longs when referencing point data ...
    ;   there can be a lot of point data in an HDF file!

    num_flashes = orbit.point.point_summary.flash_count-1L

    ; Convert the flash TAI93 times to UTC

    flash_times = NEW_DATETIME_STRUCTURE(num_flashes)
    flash_times[*].tai93 = $
                orbit.point.lightning.flash[*].tai93_time
    CALC_DATETIME, flash_times

    ; Now write to an ASCII file

    OPENW, lun, "orbit.out.f", /GET_LUN

    FOR i=0L, num_flashes-1L, 1L DO BEGIN

        PRINTF, lun, $
            flash_times[i].utc.month, $
            flash_times[i].utc.day, $
            flash_times[i].utc.year, $
```

```
            flash_times[i].utc.hour, $
            flash_times[i].utc.minute, $
            flash_times[i].utc.second, $
            orbit.point.lightning.flash[i].location[0], $
            orbit.point.lightning.flash[i].location[1], $
            orbit.point.lightning.flash[i].radiance
```

```
    ENDFOR

    CLOSE, lun
    FREE_LUN, lun

END
```

Using the C API

```
/* Sample C code to read an orbit and
   export the flashes' UTC date/time,
   latitude, longitude and radiance to
   an ASCII file */

#include <stdio.h>
#include <math.h>
#include "liblisotd_read_LISOTD.h"

main(int argc, char *argv[])
{
    struct lis_orbit TheData;
    struct anOTDBoundset Bounds;

    Initialize();
    ResetAllBounds(&Bounds);

    TheData = GetData("TRMM_LIS_SC.03.4_1998.026.00933",
                      &Bounds);

    /* UTC, lat and lon are exported by default.
       Add the radiance in manually. */

    AddASCIIOutputField(F, RADIANCE, DEFAULT);

    WriteData(TheData, ASCII_SINGLE, "orbit.out",
                                  !O, !A, F, !G, E!);
    FreeData(&TheData);

}
```

B.3 Flash rate climatology

Using the IDL API

```
/* Sample IDL code to read several
   orbits and sum them to form a
   mini-"flash rate climatology" */

@lisotd.pro
@lisapp_DateTime.pro

PRO sample3

    ; Create the grids

    flgrid = FLTARR(360)[180]
    vtgrid = FLTARR(360)[180]
    flrategrid = FLTARR(360)[180]

    ; We assume there are 32 HDF file names
    ; listed in an external ASCII file called "filelist"

    num_files = 32
    thefiles = STRARR(num_files)

    OPENR, lun, "filelist",/GET_LUN
    READF, lun, thefiles
    CLOSE, lun
    FREE_LUN, lun

    FOR n = 0, num_files-1, 1 DO BEGIN

        READ_ORBIT, thefiles[n], orbit, $
            /QUIET, /NO_AREAS, /NO_GROUPS, $
            /NO_EVENTS, /NO_ONE_SECONDS

        num_fl = orbit.point.point_summary.flash_count
        num_vt = orbit.point.point_summary.vt_count

        FOR k=0L, num_fl-1L, 1L DO BEGIN
            i = FIX(orbit.point.lightning.flash[k].location[1]+180.0)
            j = FIX(orbit.point.lightning.flash[k].location[0]+90.0)
            flgrid[i,j] = flgrid[i,j] + 1.0
        ENDFOR

        FOR k=0L, num_vt-1L, 1L DO BEGIN
```

```
            i = FIX(orbit.point.viewtime[k].location[1]+180.0)
            j = FIX(orbit.point.viewtime[k].location[0]+90.0)
            vtgrid[i,j] = vtgrid[i,j] + $
                orbit.point.viewtime[k].effective_obs
        ENDFOR

    ENDFOR

    ; The flash rate is just the flash count divided by
    ;     the effective viewtimes which are > 0

    hasbeenseen = WHERE(vtgrid gt 0, hasbeenseen_count)

    IF (hasbeenseen_ct gt 0) THEN BEGIN
        flrategrid[hasbeenseen] = $
            flgrid[hasbeenseen] / vtgrid[hasbeenseen]
    ENDIF

    ; flrategrid now contains the flash rates.  Do with it
    ;   as you please!

END
```

Using the C API

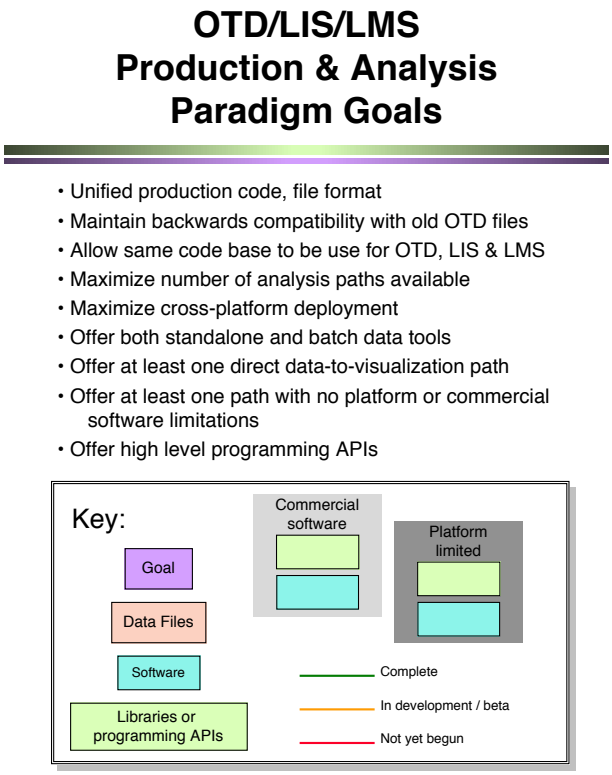
```
/* Sample C code to read several
   orbits and sum them to form a
   mini-"flash rate climatology" */

# include <stdio.h>
# include <math.h>
# include "liblisotd_read_LISOTD.h"

main(int argc, char *argv[])
{
    struct lis_orbit TheData;
    struct anOTDBoundset Bounds;
    FILE   *listfile;
    char   afilename[132];
    float  vtgrid[360][180], flgrid[360][180],
           flrategrid[360][180];
    long    i,j,k, num_fl, num_vt;

    Initialize();
    ResetAllBounds(&Bounds);

    /* Zero out the grids */

    for (i=0; i<360; i++) {
        for (j=0; j<180; j++) {
            vtgrid[i][j] = 0.0;
            flgrid[i][j] = 0.0;
            flrategrid[i][j] = 0.0;
        }
    }

    /* We assume the list of HDF file
       names is in an external ASCII file
       called "filelist" */

    listfile = fopen("filelist","r");

    while (fgets(afilename,132,listfile) != NULL) {

        TheData = GetData(afilename,&Bounds);

        num_fl = TheData.point.point_summary.flash_count;
        num_vt = TheData.point.point_summary.vt_count;
```

```
        for (k=0; k<num_fl; k++) {
            i = (int)
                (TheData.point.lightning.flash[k].location[1]+180.0);
            j = (int)
                (TheData.point.lightning.flash[k].location[0]+90.0);
            flgrid[i][j] = flgrid[i][j] + 1.0;
        }
        for (k=0; k<num_vt; k++) {
            i = (int)(TheData.point.viewtime[k].location[1]+180.0);
            j = (int)(TheData.point.viewtime[k].location[0]+90.0);
            vtgrid[i][j] = vtgrid[i][j] +
                TheData.point.viewtime[k].effective_obs;
        }

        FreeData(&TheData);

    }

    fclose(listfile);

    /* The flash rate is just the flash count divided by
        the effective viewtimes which are > 0 */

    for (i=0; i<360; i++) {
        for (j=0; j<180; j++) {
            if (vtgrid[i][j] > 0)
                flrategrid[i][j] =
                    flgrid[i][j]/vtgrid[i][j];
        }
    }

    /* flrategrid now contains the flash rates. Do with it
       as you please! */

}
```

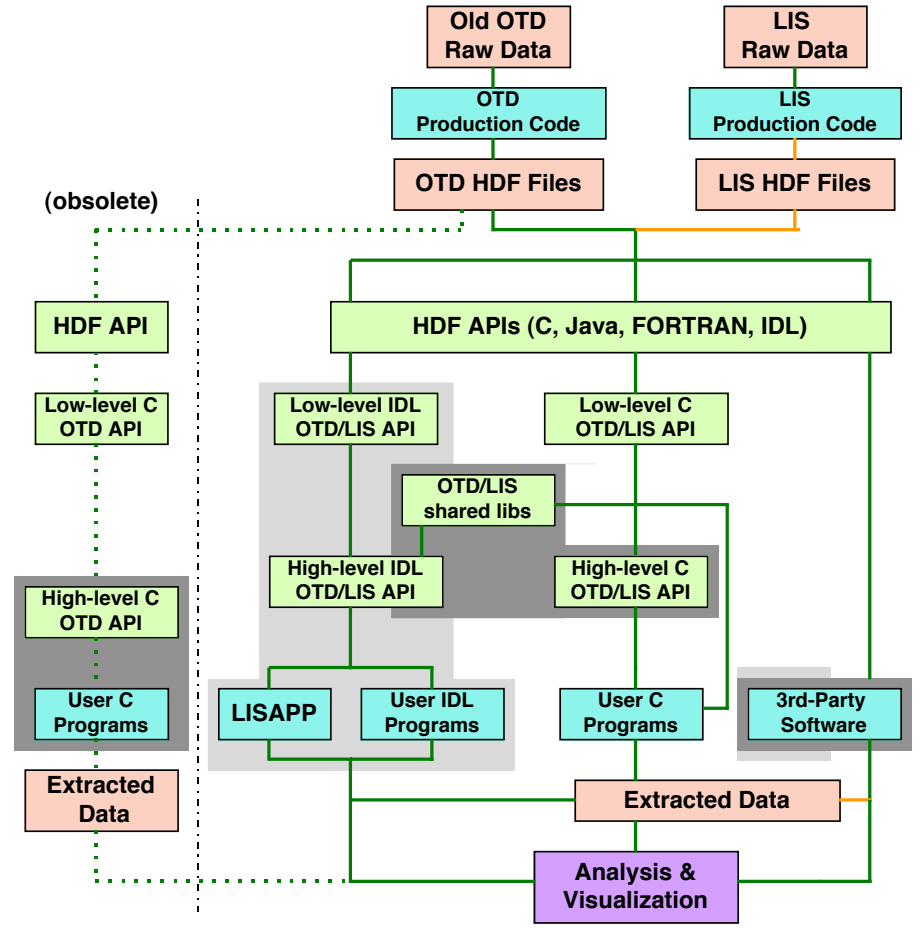
Appendix C

Software Strategy

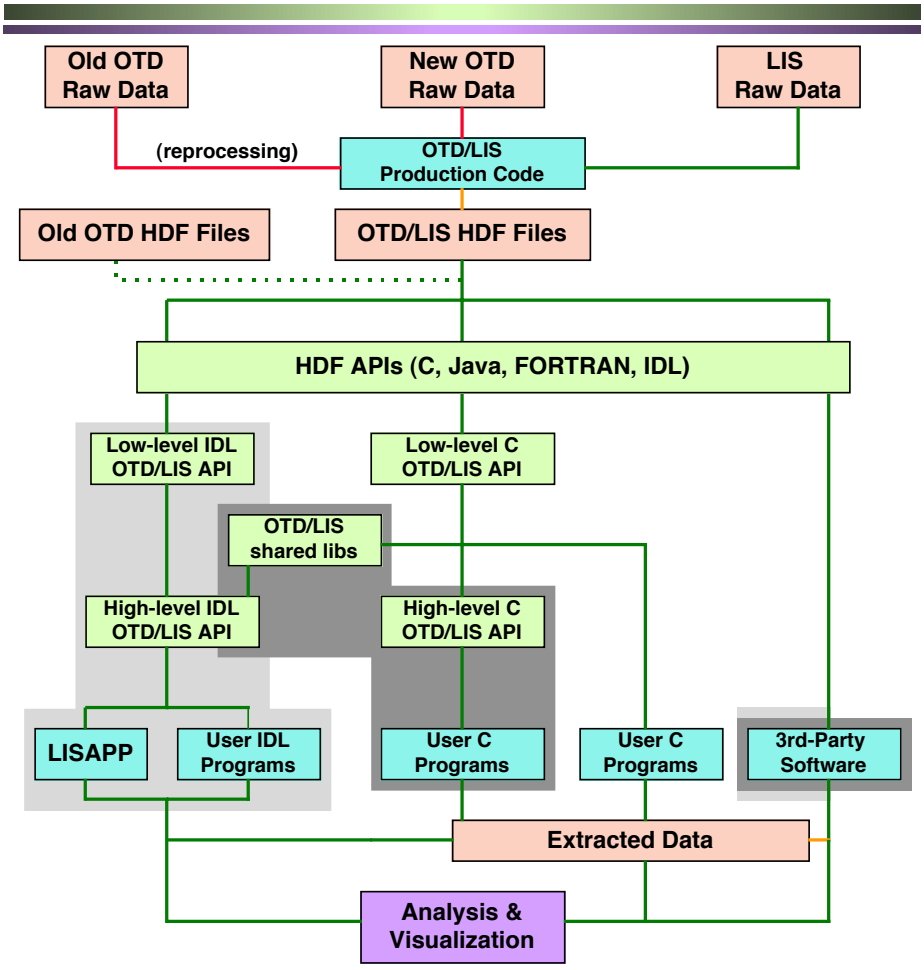
This appendix briefly summarizes the current and possible future directions of the LIS/OTD production and analysis code. The basic goals and a key to the diagrams is presented below; the roadmaps are shown for today (1998), the near future (1999) and a possible LMS scenario (2003).



Current OTD/LIS Production & Analysis Paradigm (1998)



Target OTD/LIS Production & Analysis Paradigm (1999)



Possible OTD/LIS/LMS Production & Analysis Paradigm (2003)