TITAN USERS GUIDE

Mike Dixon
Research Applications Laboratory
National Center for Atmospheric Research
Boulder Colorado USA

October 2005

TITAN Data System

Data types

The data types used by TITAN may be classified into two broad groups:

- **internal** formats, used only in systems such as TITAN;
- **external** formats, used to allow TITAN to read data from and write data to external clients.

The significant internal formats are:

- **MDV** (Meteorological Data Volume) for gridded data;
- **SPDB** (Symbolic Product Data Base) for non-gridded data such as aircraft tracks, surface station measurements, lightning, etc.;
- **TITAN** storm and track data;
- **FMQ** (File Message Queue) data for temporary storage of data in a first-in, first-out (FIFO) queue. As an example, radar beam data is handled by an FMQ.

Some examples of external formats handled by TITAN are:

- **netCDF** (NCAR gridded and non-gridded data);
- **Dorade** radar sweep (beam-by-beam) files (NCAR);
- **NEXRAD** level-2 beam-by-beam radar data;
- **RDAS** beam-by-beam radar data;
- **GRIB** (WMO gridded format).

Byte ordering

When a data type, such as a 4-byte floating point number, takes up more than 1 byte of storage, the ordering of the data bytes is important in interpreting the data.

All TITAN binary data is stored in so-called **big-endian** or **network-byte-order**. This is the native byte ordering for platforms such as SUN, and the opposite of that used by INTEL and AMD processors common to LINUX systems.

The applications which read and write the data handle the swapping of bytes to/from the big-endian format, using the libs/dataport library.

Data servers

In common usage, the word ‘server’ frequently refers to an item of hardware, as in a computer used as a server. However, in the context of this discussion, a **data server** is a program (i.e. application) which received requests from some other program (the **client**) and responds to the request by either reading data from or writing data to a local storage device (such as a hard disk).

It is helpful to understand this concept. Servers can perform a variety of tasks, such are reading data on remote machines and sending it back to the client, translating the data format on the fly, or distributing data to a remote host. The client makes requests of the server, the server handles the request and then responds to the client in an appropriate manner.

Servers generally listen on a virtual ‘port’, which is identified by a number between 0 and 32767. Port numbers below 1000 are generally reserved for use by the operating system. When a client application connects to that port, the server will create a copy of itself, either as a completely separate process or as a separate ‘thread’ of the parent process, to handle the request. That leaves the main server process or thread available to listen for other requests. (You will notice multiple copies of the same application running on the system. These copies are the child threads or processes of the servers.)

The following table lists the most common servers used in the TITAN system, including the relevant protocol and default port used.

Server name	Default port	Protocol	Remarks
procmap	5433	none	Process mapper - keeps a table of running processes
DataMapper	5434	none	Data mapper - keeps a table of data sets
DsServerMgr	5435	none	Server manager - starts servers as necessary
DsMdvServer	5440	mdvp	MDV data server
DsSpdbServer	5441	spdbp	SPDB data server
DsFmqServer	5443	fmqp	FMQ data server
DsFCopyServer	5445	fcopyp	File copy server - used by DsFileDist to distribute files
DsTitanServer	5446	titanp	TITAN data server
Ltg2Symprod	5450	spdbp	Lightning data - convert to Symprod
AcTrack2Symprod	5451	spdbp	Aircraft track data - convert to Symprod
Bdry2Symprod	5452	spdbp	Boundary (gust front) data - convert to Symprod
Metar2Symprod	5456	spdbp	Metar data - convert to Symprod
Pirep2Symprod	5457	spdbp	PIREP and AIREP data - convert to Symprod

Server name	Default port	Protocol	Remarks
Tstorms2Symprod	5460	spdbp	TITAN storm and track data - convert to Symprod
WxHazards2Symprod	5463	spdbp	Weather hazards - convert to Symprod
GenPt2Symprod	5465	spdbp	Generic point data - convert to Symprod
GenPtField2Symprod	5466	spdbp	Generic point field data - convert to Symprod
HydroStation2Symprod	5468	spdbp	Hydrology station data - convert to Symprod
SigAirMet2Symprod	5469	spdbp	SIGMETs and AIRMETS - convert to Symprod
GenPoly2Symprod	5472	spdbp	Generic polygon data - convert to Symprod
Rhi2Symprod	5473	spdbp	RHI location data - convert to Symprod

In the server list above, you will notice a number of servers with names ending in ‘2Symprod’. These are servers which translate binary data, stored in SPDB files, into a graphic object representation suitable for use by display clients. The display application `CIDD` is an example of such a client. `CIDD` needs to receive data in a graphical form because it does not understand the many types of data stored in SPDB. The graphical form of the data served to `CIDD` allows it to render the object, no matter what form it was stored as in SPDB.

Server manager

One primary data server, `DsServerMgr`, is the ‘Server Manager’. It is responsible for starting up the other servers as needed. Therefore, only the DsServerMgr need be started by TITAN. When a client needs data from a server, it will try to connect to the server. If it fails, it will then contact the server manager, which will start the server and return a message to the client once the required server is running. The client then re-tries connecting to the original server, which should then be running.

The server manager does not start `procmap` or `DataMapper`. These must be started separately.

Top level data directory, $DATA_DIR

TITAN data is stored on a host in a directory structure which reflects the type and contents of the data sets. These directories are organized below a top-level directory, referred to as the DATA_DIR.

An environment variable, `$DATA_DIR`, is set to point to the top of the directory tree.

There is a second environment variable, `$RAP_DATA_DIR`, which was used in the past and is sometimes still used. It is supported for backward-compatibility. If `$RAP_DATA_DIR` exists, it should be set to the same value as `$DATA_DIR`.

Data directory structure

The data directories lie below the top level directory, `$DATA_DIR`.

The data directories should be named in a manner which indicates (a) the format type and (b) the contents of the data set.

For example:

```
$DATA_DIR/mdv/radar/polar
$DATA_DIR/mdv/radar/cart
$DATA_DIR/mdv/clutter/cart
$DATA_DIR/mdv/sat/ir
$DATA_DIR/mdv/sat/vis
$DATA_DIR/spdb/tstorms
$DATA_DIR/spdb/ltg
$DATA_DIR/titan/storms/cart
$DATA_DIR/titan/storms/merged
$DATA_DIR/fmq/radar
$DATA_DIR/raw/sat
$DATA_DIR/grib/mm5/domain1
```

File naming conventions

For meteorological data, one of the most important attributes is **time**. Therefore many of the files are named according to the time of the data stored in the file.

A number of times may apply to the various data sets:

- valid time - the time at which an observation was made. This is also referred to as ‘observation time’;
- generate time - the time at which a model was run and forecasts generated;
- forecast time - the time at which a forecast is valid;
- lead time - the time difference between the forecast time and generate time for a forecast.

MDV file names

MDV files are named in two ways, as follows.

- By valid time: `dir/yyyymmdd/hhmmss.mdv`
- By generate time and lead time: `dir/yyyymmdd/g_hhmmss/f_llllllll.mdv`

Most data sets are stored using valid time. The files for a single day are stored in a subdirectory, which is named after the year, month and day. The files within the directory are named according to the hour, minute and second.

For forecast-style data, you can choose whether to use the valid-time naming convention, or whether to use the more complicated generate-time and lead-time convention. The former is simpler, but can lead to over-writing of data sets if forecast results from different generate times have the same valid time. For example, for a model run at 0Z and 6Z, the 9-hour forecast from the 0Z run will be over-written by the 3-hour forecast from the 6Z run.

To avoid over-writing, use the second naming convention. There are 2 levels of sub-directory. The name of the upper one is based on the year, month and day of the **generate** time. The lower one is named g_hhmmss, based on the hour, minute and second of the **generate** time.The files themselves are named using an 8-digit lead time in seconds. For example, the 6-hour forecast for the 9Z model run on 1 July 2005 will be named:

```
20050701/g_090000/f_00021600.mdv
```

since 6 hours is 21600 seconds.

SPDB file names

SPDB data resides in a simple data base, which uses time as the main key for retrieving the data (hence the name Symbolic Product Data Base.) There are 2 files for each day, one of which actually holds the data chunks and one which holds an index of the data chunks.

The files are named:

`yyyymmdd.indx` (index file)
`yyyymmdd.data` (data file)

For example, lightning data for 1 July 2005 might appear as:

```
$DATA_DIR/spdb/ltg/20050701.indx
$DATA_DIR/spdb/ltg/20050701.data
```

TITAN storm and track file names

In a manner similar to SPDB, TITAN data resides in a simple data base, which uses time as the main key for retrieving the data. There are 2 types of data, **storm** property data (location, volume, height etc.) and **track** property data (speed, growth forecast etc.). For each data type there are 2 files for each day, one of which actually holds the **data** and one which holds a **header** for the data. The file extensions have a ‘5’ at the end, which indicates TITAN data files version 5.

The files are named as follows:

`yyyymmdd.sh5` (storm header)
`yyyymmdd.sd5` (storm data)
`yyyymmdd.th5` (track header)
`yyyymmdd.td5` (track data)

For example, TITAN data for 1 July 2005 might appear as:

```
$DATA_DIR/titan/storms/20050701.sh5
$DATA_DIR/titan/storms/20050701.sd5
$DATA_DIR/titan/storms/20050701.th5
$DATA_DIR/titan/storms/20050701.td5
```

FMQ file names

File Message Queue (FMQ) files hold a queue of data. They are created once, with a predetermined size, after which the file names and sizes do not change.

The FMQ is made up of 2 files, a **status** (`.stat`) file which keeps track of the details of the queue, and a **buffer** (`.buf`) file which actually holds the data.

As an example, the following could be the name for the files of an FMQ used to handle beam-by-beam radar data:

```
$DATA_DIR/fmq/radarBeams.stat (status)
$DATA_DIR/fmq/radarBeams.buf (data buffer)
```

Note that for MDV, SPDB and TITAN data, the data set is referred to by the directory in which the data resides. For FMQ data, the root of the FMQ file name is also used, so that the FMQ example above would be referred to as '`fmq/radarBeams`'.

Data access via URL

Some TITAN applications refer directly to the **directory** in which the data resides or is to be stored. This applies to data which resides on the **local** host.

Many applications, however, refer to data which may reside either on the **local** host or on a **remote** host. In this case, we refer to the data location via a Uniform Resource Locator, or **URL**.

Most of the URLs used in TITAN are specific to the internal data system used by TITAN, and do not conform to the well-known HTTP or FTP standards. However, some URLs, specifically some of those used by the display application `CIDD`, are standard http URLs. The context will indicate when an http URL is appropriate.

A full TITAN URL has the following formal syntax:

protocol:translator:param//host:port:dir?args

or simply

dir

URL parts

The various parts of the URL are described below.

Protocol

The protocol is required.

The protocol specfies the manner in which the data will be transferred between server and client. Supported protocols are **mdvp** for MDV data, **spdbp** for SPDB data, **fmqp** for FMQ data, and **titanp** for TITAN storm track data.

Translator

The translator is optional.

This is the name of the server used to handle the request. For most URLs this will be blank, and the default application will be used. For example, for the mdvp protocol, the default application is the DsMdvServer which handles most mdv requests.

As an example of the use of a translator, consider the case of rendering lightning SPDB data on the CIDD display. The lightning data is stored in a binary format specific to lightning data. However, CIDD knows nothing about lightning data but does understand a generic graphical data format designed for this purpose. Therefore, we use the server Ltg2Symprod to transform the lightning data into the generic graphical format (Symprod) format.

param

The param is optional.

The param part identifies the parameter file to be used by the server. It is normally blank.

If an application wishes to specify that a server should use a specific parameter file to service the request, this field is filled in. For example, suppose the CIDD display wishes the Tstorms2Symprod server to use a parameter file ending in 'test', it will use a URL such as:

`spdbp:Tstorms2Synprod:test//localhost::spdb/tstorms`.

This indicates that the parameter file:

`$DATA_DIR/spdb/tstorms/_Tstorms2Symprod.test`

should be used.

host

host is required.

This is name of the host on which data resides.

port

port is optional.

This is the port on which appropriate data server is listening. It is normally left blank, in which case the standard port for that server will be used.

If a server is running on a non-standard port, the port number must be filled in.

If the host is local, many clients will try to read or write the data files directly rather than contact a server. If you need to force access to a server on the local host, set the port to 0.

dir

dir is required.

dir is the directory of the data. If it starts with '`/`' the directory is absolute, i.e. relative to the root node on the file system. If start with '`.`', the directory is relative to the directory in which the application was started. If neither of these cases is true, the directory is relative to `$DATA_DIR`.

Note that for FMQ data the dir also includes the root of the file name. See page 5 for details on the file naming convention.

args

args is optional. args are mostly left blank.

args are application-specific arguments to the URL. Both the server and client need to understand the arguments, they are not defined by the URL structure.

`?` separates the args from the directory.

The args are key-value pairs, with `=` between the key and value, and `&` between the args.

Simple directory-only URL

The simple **dir** form of the URL implies that the protocol is known from the data type, the translator and param are not specified, the host is localhost, the port is the default port and there are no args.

URL examples

The URL mechanism is powerful and it takes a while to get used to how to set up URLs. Examples are a good way to illustrate the uses.

mdv/radar/cart:

The is an example of the simplest URL, in which only the directory is specified and all of the other fields and syntax are omitted. It is equivalent to **mdvp:://localhost::mdv/radar/cart**. For most applications data access will be directly to the file, and no server will be used.Since the directory does not start with `/` or `.`, the data will be searched for in `$DATA_DIR/mdv/radar/cart`.

mdvp:://localhost:0:mdv/radar/cart:

Accesses to the same data as the simple URL above, but force the use of the DsMdvServer.

spdbp:://localhost::/tmp/spdb/ltg:

Read SPDB data from `/tmp/spdb/ltg` on the localhost It is likely that the client will read/write the files directly rather than going via a server.

Note that because the directory begins with /, it is taken as absolute and not relative to `$DATA_DIR`.

This URL could be replaced with the simple URL **/tmp/spdb/ltg**.

fmqp:://venus::fmq/dsRadar :

Request for access to the FMQ called `$DATA_DIR/fmq/radar`, on host venus.The actual FMQ files will be:

```
$DATA_DIR/fmq/radar.stat
$DATA_DIR/fmq/radar.buf
```

titanp:://mars:10000:titan/storms/merged :

Request access to the TITAN data on host mars, using the DsTitanServer listening on port 10000, and find the data in `$DATA_DIR/titan/storms/merged`.

spdbp:Ltg2Symprod:special//pluto::spdb/ltg :

Request lightning data from host pluto, reading from directory `$DATA_DIR/spdb/ltg`. Contact the server Ltg2Symprod on pluto for the data. The server must read the parameter file `$DATA_DIR/spdb/ltg/_Ltg2Symprod.special` before converting the data to Symprod format and sending it back to the client.

spdbp::distribute//localhost::spdb/ac_posn :

Accesses the Spdb server on the localhost, after it has read the file `$DATA_DIR/spdb/ac_posn/_DsSpdbServer.distruibute`. A URL of this type is often used to write data to a server, which will then distribute the data to other hosts specified in the parameter file.

MDV format for gridded data

The Meteorological Data Volume (MDV) format for gridded data was developed at NCAR in the early 1990s. At the time a number of formats were in use at RAP for similar data types. Therefore it was decided to simplify the systems by standardizing on a single data type.

No public data standard available at the time was considered suitable in terms of data encapsulation and internal compression. MDV evolved as a data format unique to NCAR. It is an efficient format for gridded data, with good meta-data support and an efficient internal compression capability which allows for selected decompression of a single plane from a single data field.

Data formats such as GRIB and HDF perform a similar function to MDV. To the extent possible, converters are being developed to allow transformation between MDV and other commonly-available data formats.

See page 4 for details on MDV file naming conventions.

Mdv concepts

MDV is a general purpose data file format for storing one-, two- and three-dimensional gridded data.

The MDV file format is highly structured and provides capabilities for managing multiple data fields in a single file. For example, one MDV file might contain a data volume with three tilts of radar reflectivity and radial velocity, along with a model-generated volume of wind vectors at these three tilts.

MDV requires constant spacing of data in the x-y plane of each field, i.e. a single delta-x and delta-y for all data in each field. However, delta-x and delta-y may vary from field to field. In the third dimension, MDV supports both variable data spacing, with a maximum of 122 vertical levels.

A simple example of a radar MDV file is one with two data fields, reflectivity and velocity, where (x,y,z) represents (range, azimuth, elevation) and where:

Table 1—

field	**field**	**num**	**num**	**num**	**min**	**min**	**min**	**delta**	**delta**	**delta**
name	units	x	y	z	x	y	z	x	y	z
DBZ	dBZ	600	360	2	0.375	0.0	0.0	0.25	1.0	1.0
VEL	m/s	600	360	2	0.375	0.0	0.0	0.25	1.0	1.0

In this example, a look into a few of the headers fields might reveal:

```
MasterHeader.vlevel_included: true
FieldHeader(DBZ).proj_type: Radar Polar
VlevelHeader(DBZ).vlevel_type[0]: Elevation Angle
VlevelHeader(DBZ).vlevel_type[0]: Elevation Angle
VlevelHeader(DBZ).vlevel_params[0]: 0.5
VlevelHeader(DBZ).vlevel_params[1]: 1.45
```

Units for x and y are defined implicitly by the field's projection type. In this example, the "Radar Polar" projection type prescribes units of km for x (range), deg for y (azimuth). The units for z are defined implicitly by the field's vlevel type. In this example, vlevel headers are included so the type is taken from the vlevel header for each level and the "Elevation Angle" vlevel type prescribes units of deg for z. The vlevel_params array gives the elevation angle for each vertical level.

The MDV file format is extensible in that it provides space and access capabilities for generic "chunk" data defined by the MDV user. Chunk data allows MDV users to attach to their data sets additional information that is not already stored in the MDV header information. Examples of chunk data might be detailed data collection information such as satellite or radar meta-data.

In addition to the defined MDV file format, a C++ library (libs/Mdv) supports an application programmer's interface (API) to provide high-level facilities for reading and writing MDV files.

Other features of the MDV file format and library API include:

- a FORTRAN-compatible I/O structure (for non-compressed data);
- date/time stamping for creation, expiration, and forecasting;
- support for data compression/decompression;
- byte swapping to handle cross-platform data representation schemes.

Overview of the MDV file format

The MDV FILE STRUCTURE looks like the following:

```
MDV_master_header
MDV_field_header 1
MDV_field_header 2 ...
MDV_field_header n
MDV_vlevel_header 1 (optional)
MDV_vlevel_header 2 ...
MDV_vlevel_header n
MDV_chunk_header 1 (optional)
MDV_chunk_header 2 ...
MDV_chunk_header n
field 1 data
field 2 data ...
field n data
chunk 1 (optional) ...
chunk n
```

All MDV header information appears at the beginning of the file followed by the field data and any chunk data. The field headers and chunk headers provide data offsets from the start of the file to the field or chunk data.

MDV Headers

A master header is followed by a field header for each of the fields in the data set. The master header and field headers are required.

Although vlevel headers are technically optional, all C++-generated mdv files have them. Vlevel headers are necessary for data with variable spacing or variable units in the third dimension. Vlevel headers are used to store the details of the third dimension data, e.g. radar elevation angles. For data sets with constant spacing and units in the third dimension, the master header and/or field headers contain sufficient information for interpreting the data.

Although the vlevel headers are optional, if one vlevel header exists for a field, then a vlevel header must exist for each field in the data set. A flag in the master header indicates whether or not vlevel headers are included in the data set.

Following the vlevel headers are optional chunk headers. The structure and content of the chunk data is defined by the application programmer. Thus, interpreting chunk data requires “a-priori” knowledge of the structure and contents of the chunk.

Because MDV is built upon an in-house portability library (dataport), data types within MDV headers are specified using dataport definitions rather than compiler-specific data types. The following dataport types are used in the MDV headers:

```
typedef signed char si08; // 1 byte
typedef unsigned char ui08; // 1 byte
typedef signed short si16; // 2 bytes
typedef unsigned short ui16; // 2 bytes
typedef signed int si32; // 4 bytes
typedef unsigned int ui32; // 4 bytes
typedef float fl32; // 4 bytes
```

For the definitions of the MDV file headers, see the include file `include/Mdv/Mdv_typedefs.h`.

MDV Field Data

Data records for each MDV field follow all header information.

The MDV file format has been designed to support various data types for internal storage (bytes, shorts, floats, RGBA). The following data portability types are supported in MDV field data:

```
typedef unsigned char ui08; // 1 byte
typedef unsigned short ui16; // 2 bytes
typedef float fl32; // 4 bytes
typedef unsigned int ui32; // 4 bytes used for RGBA
```

In order to represent floating point data as single bytes or short integers, MDV provides a bias and scaling factor in each field header.

MDV data may be stored internally in uncompressed or compressed form. A number of compression schemes are supported, including 1-byte run-length encoding (RLE8), GZIP and BZIP2. The data is compressed on a plane-by-plane basis, so that a single plane from a single field may be decompressed and used while leaving the remained of the file compressed. This makes specific data retrieval more efficient.

For details on supported MDV data compression types, see the include file `include/Mdv/Mdv_enums.h`.

SPDB data base

The SPDB (Symbolic Product data base) format was developed for simple and efficient time-referenced storage of a wide range of non-gridded meteorological data. It is used in TITAN for almost all data which does not readily lend itself to storage in a regular grid.

Examples of data stored in SPDB are lightning data, aircraft tracks, aircraft measurements, gust fronts, surface station measurements and hydrological measurements.

SPDB stores its data in ‘day files’, two files per day. See page 5 for details on the file naming convention.

The .indx file contains information on how the individual data chunks are stored, as well as some special time-index arrays to speed up data access. All headers in the index file are stored out in big-endian byte ordering by SPDB.

The .data file contains the actual data chunks stored in the data base.The SPDB sub-system does not attempt to interpret the data in any way. It assumes that the data chunk is in big-endian format when passed to it, and it stores the data exactly as provided by the client.

Programming access to SPDB files is performed using the C++ API in the libs/Spdb library.

TITAN storms and track data

The TITAN storm and track data is in ‘day files’, four files per day. See page 5 for details on the file naming convention.

Generally, the files contain data for a single data, plus an **overlap-period** which is specified in the Titan application parameter file.

Programming access to Titan files is performed using the C/C++ API in the libs/titan library.

FMQ - File Message Queues

Certain data types are best handled by first-in, first-out (FIFO) queues. A good example of this is radar beam-by-beam data, which has a high data rate and which must be buffered for the applications which use it.

The FMQ library (lib/Fmq) implements the file message queue mechanism. Two files are created per queue, one for the status of the queue (.stat) and one for the actual data buffered up by the queue (.buf). See page 5 for details on the file naming convention.

Just as with SPDB, the FMQ does not attempt to interpret the data chunks stored in the buffer, it merely handles them as a black box and stores them for later use by the reader. The writer and reader of the data must agree mutually on the format of the stored data. The FMQ expects the data to be in big-endian format when passed to it for writing.

Latest data information files

In a real-time system which must respond to data as it arrives, considerable effort is placed on knowing when new data is available.

A common mechanism employed is the scanning of incoming data directories to search for new files as they arrive. This method is employed in TITAN where raw data arrives from outside the system.

This method has the disadvantage that it can be quite costly in terms of CPU usage when a directory contains a large number of files. Also, it is often necessary to monitor sub-directories for data arrival, making it more complicated and expensive. Therefore, where possible, TITAN uses the latest-data-info file mechanism to notify clients of data arrival.

The idea is simple. When an upstream application writes data to a directory, it also writes a simple ASCII file in the data-set directory. The downstream application then merely has to monitor that one file, rather than the entire directory and/or sub-directories.

An important point here is that the time in the latest_data_info file is the time of the data most recently written to disk. So if you are running in archive mode, and the data you are using is old, the latest data time will be in the past.

In practice, 4 files are written rather than one. These files are:

```
_latest_data_info
_latest_data_info.xml
_latest_data_info.stat
_latest_data_info.buf
```

_latest_data_info.xml

The XML file is the best one to look at in terms of understanding what is going on. Here is an example of an _latest_data_info.xml file:

```
<latest_data_info>
  <unix_time>1089774255</unix_time>
  <year>2004</year>
  <month>07</month>
  <day>14</day>
  <hour>03</hour>
  <min>04</min>
  <sec>15</sec>
  <rel_data_path>20040714/030415.mdv</rel_data_path>
  <file_ext>mdv</file_ext>
  <data_type></data_type>
  <user_info1>none</user_info1>
  <user_info2>none</user_info2>
  <is_forecast>false</is_forecast>
  <forecast_lead_secs>0</forecast_lead_secs>
```

```
    <writer>MdvMerge2</writer>
  </latest_data_info>
```

The various xml fields are:

- `unix_time`: the number of seconds since 1 January 1970;
- `rel_data_path`: path of the actual data file written, relative to the data set directory, i.e. the directory containing the _latest_data_info file;
- `file_ext`: extension of the data file;
- `data_type`: mdv, spdb, titan, raw, etc.
- `user_info_1`: available for client data, the reader and writer must agree on interpretation of this field;
- `user_info_2`: available for client data, the reader and writer must agree on interpretation of this field;
- `is_forecast`: flag set 0 if the data is not for a forecast, 1 if it is;
- `forecast_lead_secs`: lead time for data file, if applicable. For forecast data, the time in the latest_data_info file represents the generate time. The valid time for the forecast is computed by adding the lead_secs to the generate time;
- `writer`: name of the application which wrote the data.

Some older applications used the `user_info_1` field for '`writer`' and `user_info_2` as '`rel_file_path`'. Therefore you might see some duplicated use of the fields in the file. This will be cleaned up with new software releases.

_latest_data_info

The _latest_data_info file, without any extension, is there for backward compatibility purposes only, and will eventually be removed.

The following is an example of a `_latest_data_info` file. It corresponds to the XML file example in the section above.

```
1089774255 2004 7 14 3 4 15
mdv
MdvMerge2
20040714/030415.mdv
0
```

Line 1 holds the date and time. Line 2 holds `file_ext`. Lines 3 and 4 hold `user_info_1` and `user_info_2`. Line 5 holds the `is_forecast` flag. If `is_forecast` is 1, there will be line 6 which will hold the `forecast_lead_time`.

_latest_data_info FMQ

You might recognize the following 2 files as belonging to an FMQ.

```
_latest_data_info.stat
_latest_data_info.buf
```

In addition to writing the xml file, the latest data info is also written to an FMQ which by default has a queue size of 1024. The reason for doing this is that sometimes data arrives very fast, more than once per second. Typically the latest data info file is scanned once per second. If we were to rely on the XML file, and data arrives faster than once per second, we would miss data as it arrives.

The FMQ takes care of that problem. The client reads the queue instead of the XML file, and can therefore catch-up by reading a number of queue entries one after the other to respond to high-rate data.