TITAN USERS GUIDE
Running TITAN

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

TITAN comprises a large number of applications ranging in purpose from data ingest to algorithms to display and data visualization. Most of these applications are designed to be run in two modes: REALTIME mode, in which data arrives in real-time and is stored and analyzed as it arrives, and ARCHIVE mode, in which analysis is performed on some part of the data set some time after the data is collected.

Many of the TITAN applications may be thought of as simple steps in a more complex system. The applications read data in one form, perform some type of procedure or algorithm on it and write it out on another form. The next process in the chain will then read the data output from the previous step, perform some action and write the results out ready for subsequent steps, and so on. In this way we can break up a complex processing system into a number of smaller, easy-to-handle steps.

Running TITAN applications

Since TITAN runs under UNIX, the TITAN applications are started by issuing commands either in a text window or via a script. (See the UNIX Basics section for an introduction to UNIX.).

Most of the TITAN applications are configured using a file containing parameters specific to the application, and commonly referred to as the 'param file'. Some simple applications do not use a parameter file and are configured entirely using command-line arguments. For more on command line arguments, see the Unix Basics section of this Users Guide.

Command line arguments

Almost all TITAN applications conform to three conventions on command line arguments, '`-h`', '`-print-params`' and '`-params`'.

- `-h`: get help, or print the usage.
- `-params`: specifies which parameter file to use for running the application.
- `-print_params`: print a default version of the parameter file to stdout.

For example, the following is the following is the result of running '`Titan -h`':

```
Usage: Titan [options as below]
options:
       [ --, -h, -help, -man ] produce this list.
       [ -debug ] print debug messages
       [ -end "yyyy mm dd hh mm ss"] end time
          ARCHIVE and RETRACK modes only
       [ -mode ?] ARCHIVE, REALTIME or RETRACK
       [ -start "yyyy mm dd hh mm ss"] start time
          ARCHIVE and RETRACK modes only
       [ -verbose ] print verbose debug messages
NOTE: for ARCHIVE mode and retracking, you must specify the analysis times
using -start and -end.
TDRP args: [options as below]
   [ -params path ] specify params file path
   [ -check_params] check which params are not set
   [ -print_params [mode]] print parameters
     using following modes, default mode is 'norm'
       short:   main comments only, no help or descr
                structs and arrays on a single line
       norm:    short + descriptions and help
       long:    norm  + arrays and structs expanded
       verbose: long  + private params included
       short_expand:   short with env vars expanded
       norm_expand:    norm with env vars expanded
       long_expand:    long with env vars expanded
       verbose_expand: verbose with env vars expanded
   [ -tdrp_debug] debugging prints for tdrp
   [ -tdrp_usage] print this usage
```

Note that the argument list is split into two parts. The top part lists the arguments for that specific application. The lower part lists the arguments related to the parameter file. TDRP stands for 'Table Driven Runtime Parameters'. Almost all TITAN applications use TDRP for their parameters.

The 'Titan' application is the program which performs the storm identification and tracking function for the TITAN system. As you can see from the above example, you can run Titan in REALTIME or ARCHIVE mode. In ARCHIVE mode, you need to specify the start and end time for the analysis.

In REALTIME mode, you might start TITAN as follows:

```
Titan -params Titan.test -mode REALTIME >& /tmp/log.Titan &
```

Note that the '>&' re-directs the output from Titan to the log file. This assumes you are starting it up from with the C shell. In REALTIME mode it is important to re-direct output from all

applications to a file, otherwise the application may stall trying to write the output to a terminal.

In ARCHIVE mode, you might start TITAN as follows:

```
Titan -params Titan.test -mode ARCHIVE \
-start "2005 09 21 00 00 00" -end "2005 09 22 00 00 00"
```

Parameter files

As mentioned above, almost all TITAN applications use parameter files for their configuration details. The parameter file to be used is specified on the command line, using the '`-params`' command line argument.

As an example application, we will use Dsr2Vol. This application reads beam-by-beam radar data, in radar polar coordinates, interpolates the data and converts it into MDV gridded file format. Dsr2Vol reads its input data from a file message queue (FMQ).

`Dsr2Vol -h` gives the following output:

```
Usage: Dsr2Vol [options as below]
options:
       [ --, -h, -help, -man ] produce this list.
       [ -debug ] print debug messages
       [ -verbose ] print verbose debug messages
TDRP args: [options as below]
   [ -params path ] specify params file path
   [ -check_params] check which params are not set
   [ -print_params [mode]] print parameters
     using following modes, default mode is 'norm'
       short:   main comments only, no help or descr
                structs and arrays on a single line
       norm:    short + descriptions and help
       long:    norm  + arrays and structs expanded
       verbose: long  + private params included
       short_expand:   short with env vars expanded
       norm_expand:    norm with env vars expanded
       long_expand:    long with env vars expanded
       verbose_expand: verbose with env vars expanded
   [ -tdrp_debug] debugging prints for tdrp
   [ -tdrp_usage] print this usage
```

We can see from this that Dsr2Vol uses TDRP parameters. You can view the default parameters for the application by running it with the `-print_params` command line argument.The following is a truncated version of the output from `-print_params`:

```
/**************************************************************
 * TDRP params for Dsr2Vol
 **************************************************************/
//==============================================================
//
// Dsr2Vol program.
//
// Dsr2Volreads an input FMQ containing radar data, and writes it to a
//   file in MDV format. Grid remapping and spatial interpolation are
//   optional
//==============================================================
//==============================================================
//
// DEBUGGING AND PROCESS CONTROL.
//
//==============================================================
///////////// debug ////////////////////////////////
//
// Debug option.
// If set, debug messages will be printed appropriately.
//
// Type: enum
// Options:
//     DEBUG_OFF
//     DEBUG_NORM
//     DEBUG_VERBOSE
//

debug = DEBUG_OFF;

///////////// instance ////////////////////////////////
//
// Process instance.
// Used for registration with procmap.
// Type: string
//

instance = “Test”;

//==============================================================
//
// DATA INPUT.
//
//==============================================================

///////////// input_fmq_url ////////////////////////////
```

```
//
// Input URL for DsRadar data via FMQ.
// Type: string
//

input_fmq_url = "fmqp:://localhost::./fmq.dsRadar";

// etc.
```

At the start of the parameter file is a short description of what the application does. The goal is that all TITAN applications will have a description of this sort, although at present many do not.

The parameter list contains comments in the style of the C++ language. These are either a region starting with '`/*`' and ending with '`*/`', or a region on a line following '`//`'.

The parameters are specified as key-value pairs, followed by a ',' or ';'. String values such as file names and URLs must be quoted. There is complete TDRP documentation on-line at the TITAN web site:

```
www.ral.ucar.edu/projects/titan/tdrp
```

Creating and updating parameter files

We can create a parameter file for the application by running:

```
Dsr2Vol -print_params > Dsr2Vol.example
```

This will create a parameter file `Dsr2Vol.example`, which has the default parameters set. You can then edit the parameters to suit your needs. The parameter file is intended to be self-commenting, so that by reading the comments in the file you should be able to determine how to set the parameters.

Sometimes, complicated parameters which are arrays of lists become hard to read. In these cases it is helpful to produce a '`long`' listing for the parameter file. You would do this as follows:

```
Dsr2Vol -print_params long > Dsr2Vol.example
```

The following shows an example of this. We have an array of output fields, which looks like this in the short printout:

```
output_fields = {
  { "DBZ", "DBZ", "dBZ", "none", TRUE, FALSE, FALSE, ENCODING_INT8},
  { "VEL", "VEL", "m/s", "none", FALSE, FALSE, TRUE, ENCODING_INT8}
};
```

If, instead, we use the -long option, the parameters will be printed as follows:

```
output_fields = {
  {
    dsr_name = "DBZ",
    output_name = "DBZ",
    output_units = "dBZ",
```

```
      transform = “none”,
      is_dbz = TRUE,
      interp_db_as_power = FALSE,
      is_vel = FALSE,
        encoding = ENCODING_INT8
    }
    ,
    {
      dsr_name = “VEL”,
      output_name = “VEL”,
      output_units = “m/s”,
      transform = “none”,
      is_dbz = FALSE,
      interp_db_as_power = FALSE,
      is_vel = TRUE,
      encoding = ENCODING_INT8
    }
};
```

You can also use an old parameter file to create a new, updated one. This is useful when you have a new version of TITAN which uses parameters which are not listed in the old file.

The following is an example:

```
Dsr2Vol -params Dsr2Vol.old -print_params > Dsr2Vol.new
```

NOTE: you cannot re-direct the output from one param file into the same file name, all in one step. This WILL DESTROY your original parameter file and leave you with nothing. If you want to re-use the parameter file name, you need to use a temporary file, as the following example shows:

```
Dsr2Vol -params Dsr2Vol.good -print_params > junk
mv junk Dsr2Vol.good
```

You create a temporary file called junk and then rename it to your original file.

NOTE 2: there are a few applications which are not based on TDRP (see next section) and for which this does not work. Specifically, CIDD, Rview and TimeHist are not TDRP-based. To update an old param file you will need to cut-and-paste manually from one to the other.

Applications with non-TDRP parameters

A few TITAN applications are based on an older parameter file system. These are the display applications CIDD, Rview and TimeHist, as well as some of the older applications in TITAN which are no longer used much. CIDD actually has a mixture of TDRP and non-TDRP parameters.

The applications all support the -print_params functionality, which you can use to generate a

default parameter file. However, you cannot use this in conjunction with -params to update an old file - see section above.

The non-TDRP parameters have a different syntax derived from the resource manager of the X windows system. The following are a few examples:

```
############################################################
# Example of CIDD parameters
#
# Set to 1 to start up cidd in movie-on mode, 0 = off
cidd.movie_on: 0

# Set the delay at the end of the movie loop in msec
cidd.movie_delay: 3000

# Set the speed of the movie loop. - (msec per frame)
cidd.movie_speed_msec: 75

# If set to 1 - forces CIDD to reload all data every time the movie frames
# rotate one old frame out and generate a new frame.
cidd.reset_frames:   0


############################################################
# Example of Rview parameters
#
# Geographical extent
#
# projection may be 'flat' for local plots or 'latlon' for
# regional or larger plots.
#
# The grid origin is always in lat/lon units.
#
# min and max values are in km for flat proj,
# lat/lon for latlon proj
#

Rview.projection: flat

Rview.grid_lat: 39.787
Rview.grid_lon: -104.546
Rview.grid_rot: 0.0

Rview.full_min_x: -200.0
Rview.full_min_y: -200.0
Rview.full_max_x: 200.0
Rview.full_max_y: 200.0
```

```
##############################################################
# Example of TimeHist parameters
#
# startup plotting options
#

TimeHist.plot_thist_vol: true
TimeHist.plot_thist_area: true
TimeHist.plot_thist_pflux: true
TimeHist.plot_thist_mass: true
TimeHist.plot_thist_vil: true
TimeHist.plot_thist_forecast: false #! options: false, limited, all
TimeHist.plot_thist_fit: false
```

Follow the examples of other parameters in the files to determine how to edit the parameters.

Environment variables in parameter files

It is convenient to be able to refer to environment variables in the parameter files. This is done using a syntax similar to that used in Makefiles. The variable is referred to by name, put in parentheses and preceded by the '$' character.

For example, if we want to refer the environment variable `$DATA_DIR` in a parameter, it might look like this:

```
output_dir = $(DATA_DIR)/mdv/radar/cart;
```

`$(DATA_DIR)` will be expanded to its value at runtime.

The run-time environment

When setting up the account under which TITAN will be run, you should `tcsh` (or `csh`) as the login shell. The environment variables used to control TITAN are set up in the '`.cshrc`' file which is read by the shell.

You will find example '`.cshrc`' files in the titan distribution:

```
rap/distribs/titan/dotfiles/cshrc
```

This file shows examples of how set up the path, environment variables and aliases.

Run-time environment variables

The following important environment variables are set in directly in '.cshrc':

```
setenv PRINTER lp
```

```
setenv PROJ_DIR $HOME/projDir
setenv TITAN_HOME $PROJ_DIR
setenv DATA_DIR $PROJ_DIR/data
setenv RAP_DATA_DIR $DATA_DIR
setenv PROCMAP_HOST localhost
setenv DATA_MAPPER_ACTIVE true
setenv LDATA_FMQ_ACTIVE true
setenv ERRORS_LOG_DIR $PROJ_DIR/logs/errors
setenv RESTART_LOG_DIR $PROJ_DIR/logs/restart
setenv DS_COMM_TIMEOUT_MSECS 60000
```

Note that:

- `$DATA_DIR` and `$RAP_DATA_DIR` are set equal;
- `$PROCMAP_HOST` tells applications to register with the local procmap;
- `$DATA_MAPPER_ACTIVE` tells applications to register with the DataMapper;
- `$LDATA_FMQ_ACTIVE` tells applications to write the FMQ associated with the _latest_data_info files;
- `$ERRORS_LOG_DIR` and `$RESTART_LOG_DIR` set the log directories;
- `$DS_COMM_TIMEOUT_MSECS` sets the millisecond value for time-outs between applications and the data servers.

Site-specific environment variables

The `template_single_radar, template_tseries` and `template_rdas2k` templates are both intended for use with a single radar. In order to simplify the configuration, most of the parameters which are specific to a radar site are set as environment variables, so that the individual parameter files can be set up to refer to these variables and do not need to be altered from site to site.

For example, the site-specific environment variables for the single radar template may be found in the file:

```
rap/projects/titan/templates/template_single_radar/system/params/site_params
```

After installation of the project, the file will be:

```
~/projDir/system/params/site_params
```

The file contains the following c-shell commands:

```
# Host where the data is stored
setenv DATA_HOST                  localhost
# RDAS communications
setenv RDAS_HOST                  rdas
setenv RDAS_PORT                  10000
# Radar details
setenv RADAR_NAME                 FTG
setenv RADAR_DESCRIPTION          "Front Range NEXRAD"
setenv RADAR_DATA_INFO            "FTG radar, Denver, Colorado"
setenv RADAR_LAT                  39.787
setenv RADAR_LON                  -104.546
setenv RADAR_ALT                  1.71
setenv RADAR_CONSTANT             -157.0
setenv RADAR_WAVELENGTH           5.0
setenv RADAR_BEAM_WIDTH           1.6
setenv RADAR_NOISE_DBZ_AT_100KM   -9.0
# time between volumes (secs)
setenv SCAN_DELTA_T               240
# Lowest cappi for cart grid (km)
setenv CART_GRID_START_HT         1.0
# Start height for Rview
setenv RVIEW_START_HT             2.0
# Map config file for Rview
setenv RVIEW_MAP_CONF_FILE        denver.conf
# Precip estimation
setenv ZR_COEFF                   200.0
setenv ZR_EXPON                   1.6
setenv HAIL_DBZ_THRESHOLD         53
# Directory for cart radar data
# If ClutterRemove is active, use cart_no_clutter
# If ClutterRemove is no active, use cart
setenv RADAR_CART_DIR             mdv/radar/cart
#setenv RADAR_CART_DIR            mdv/radar/cart_no_clutter
```

This file is '`sourced`' by the c-shell when it reads the '`.cshrc`' file. All this means is that the commands in the file are executed, which results in the various environment variables being set accordingly.

These site-specific parameters are referred to in many of the parameter files in the templates.

NOTE: you do not need to use these. This is only a suggestion on how things might be set up. You are free to ignore the environment variables and put the values you use directly in the parameter file, and in fact this is by far the most common practice. Also, you can of course define your own environment variables and use them too.

Shell aliases

A number of shell aliases are defined in the '.cshrc' file. An alias can be used as a short-hand for a more complex command.

```
alias setprompt 'set prompt="('hostname') 'basename $cwd' ! % "'
alias cd 'cd \!*;setprompt; echo $cwd'
alias ls 'ls -F'
alias dir 'ls -lgF \!* | more'
set history=100
alias h history 40

alias rm 'rm -i'
alias mv 'mv -i'
alias cp 'cp -i'

alias df 'df -k'

alias catw 'cat 'which \!*''
alias lessw 'less 'which \!*''

alias ensc 'enscript -2r -fCourier7'

alias ppm 'print_procmap -hb -up -status'
alias pdm 'PrintDataMap -all -relt -lreg'
alias pcheck \
  'procmap_list_check -proc_list ~/projDir/control/proc_list'
```

Note that:

- The `setprompt` and `cd` aliases format the prompt which appears in terminal windows.
- The `rm`, `mv` and `cp` aliases prompt you before deleting or copying over files.
- `catw` is a useful alias - it allows you to list a script without knowing it's location. For example, '`catw snuff`' will find the `snuff` script and list it. `lessw` is similar except it pipes the output through `less`.
- `ensc` is useful for printing.
- `ppm`, `pdm` and `pcheck` are useful for monitoring the system.

The search path

The search path is a series of file paths which are searched for when an application or script is to be run. The path is created in the '`.cshrc`' file as follows:

```
# --- Path ---
#
# Assemble path to include all scripts directories
# then projDir/bin, then the rest

set path = (.)
foreach dir ($PROJ_DIR/*)
  if (-d $dir/scripts) then
    set path = ($path $dir/scripts)
  endif
end
set path = ($path $RAP_INST_BIN_DIR ~/bin \
          $RAP_BIN_DIR $JAVA_HOME/bin \
          /usr/java/bin /usr/local/bin /usr/local/sbin \
          /usr/X11R6/bin /usr/bin /usr/sbin /bin /sbin )
```

First, all the script directories (`projDir/system/scripts`, `projDir/ingest/scripts` etc.) are added, followed by the directory will the application binaries (`~/rap/bin`) and then followed by the rest of the system directories.

Remember that if you add an application or script to the directories in the search path, you must issue the c-shell command '`rehash`' before the shell will find the new file.

The real-time system

The TITAN real-time system has a number of key components which work together to start and stop the system, keep it running, monitor it, keep the disk from filling up, log errors and so on. These details of these components will be presented in this section.

Real-time system directory structure

For a real-time installation of TITAN, there exists a top-level directory, normally called `projDir`, and referred to by the environment variable `$PROJ_DIR`. The directories for the project lie below projDir. Often the data and log directories are on a separate disk partition, because of disk usage requirements for the data, in which case these directories will be symbolic links.

The following lists a typical directory structure for a TITAN system running in real-time:

```
rap/bin    (binaries and general scripts)
projDir/
  control/ (process list and cron table)
  data/    (probably a symbolic link to a data disk)
    raw/   (raw input data)
    mdv/   (MDV data)
    spdb/  (SPDB data)
```

```
  fmq/   (FMQ data)
  titan/ (TITAN storm track data)
logs/      (may be a symbolic link)
  errors/  (error logs)
  restart/ (restart logs)
system/
  scripts/  (general system scripts)
  params/   (general system parameters)
ingest/
  scripts/  (start scripts for ingest processes)
  params/   (parameters for ingest processes)
titan/
  scripts/  (start scripts for titan processes)
  params/   (parameters for titan processes)
display/
  scripts/  (start scripts for displays)
  params/   (parameters for displays)
  maps/     (maps for displays)
  color_scales/ (color scales for displays)
```

Real-time system components

Scripts and binaries

All of the application binaries and some system scripts are found in `~/rap/bin`. These are the programs which actually perform the work in the real-time system. In addition to the scripts in the bin directory, the start scripts for the processes are found in the script directories in the `system`, `ingest`, `display` etc. sub-directories.

Control files

There are two main control files, in `projDir/control.` These are:

- `proc_list:` the list of processes which should run;
- `crontab`: the cron table which is installed when the system starts. cron is a system service which runs tasks on a schedule. The crontab specifies the tasks to be run.

Sometimes these will be symbolic links, as in template_tseries.

Process list file

The process list controls which applications (also called processes) should be run in the system. It specifies the process name, the instance and the start and kill scripts for the process. The hostname is included for backward compatibility and should always be set to localhost.

The following is an example of a proc_list file, taken from the TITAN project templates `template_single_radar`:

```
##############################################################################
# Example proc_list file
#
# name       instance   start_script          kill_script       hostname
##############################################################################
# SYSTEM processes
#
DsServerMgr   primary    start_DsServerMgr    snuff_inst        localhost
Janitor       primary    start_Janitor        kill_Janitor      localhost
Scout         primary    start_Scout          kill_Scout        localhost
DataMapper    primary    start_DataMapper     kill_DataMapper   localhost
##############################################################################
# INGEST processes
#
Bprp2Dsr       ops       start_Bprp2Dsr.ops        snuff_inst  localhost
EsdAcIngest    ops       start_EsdAcIngest.ops     snuff_inst  localhost
Dsr2Vol        ops       start_Dsr2Vol.ops         snuff_inst  localhost
ClutterRemove  cart      start_ClutterRemove.cart  snuff_inst  localhost
##############################################################################
# TITAN ROCESSES
#
Titan          ops       start_Titan.ops           snuff_inst  localhost
PrecipAccum    single    start_PrecipAccum.single  snuff_inst  localhost
PrecipAccum    1hr       start_PrecipAccum.1hr     snuff_inst  localhost
PrecipAccum    24hr      start_PrecipAccum.24hr    snuff_inst  localhost
Mdv2Vil        ops       start_Mdv2Vil.ops         snuff_inst  localhost
Tstorms2Spdb   ops       start_Tstorms2Spdb.ops    snuff_inst  localhost
##############################################################################
# DISPLAY processes
#
Rview          ops       start_Rview.ops      snuff_inst       localhost
TimeHist       ops       start_Rview.ops      snuff_inst       localhost
RadMon         ops       start_RadMon.ops     kill_RadMon.ops  localhost
CIDD           ops       start_CIDD.ops       snuff_inst       localhost
```

The process binary must be in the search path. The instance for a process is used to distinguish between difference instances of the same process. In the example above, `PrecipAccum` is running with 3 different instances, one to convert single radar scans into precipitation amount and the other two to accumulate precipitation into 1 and 24 hour running totals.

The start script for the processes should always exist. Is special action must be taken to kill the application, a kill script should also be supplied. However, if nothing special is needed to

kill the application the entry ‘snuff_inst’ can be used instead. Based on that entry the system will kill the application based on its name and instance.

Cron table file

The cron daemon on a UNIX system is designed to run tasks on a pre-defined schedule. The so-called cron table is used to specify which tasks are to be scheduled.

Below is a typical crontab file for a TITAN real-time system:

```
#########################################################################
#
# Example cron table for TITAN
#
# Process restarters
*/1 * * * *  csh -c “start_auto_restart_check_cron” 1> /dev/null 2>
/dev/null
*/1 * * * *  csh -c “start_procmap_check_cron”      1> /dev/null 2>
/dev/null
#
# Build links to log date subdirs
*/5 * * * *   csh -c “start_build_logdir_links” 1> /dev/null 2> /dev/null
#
```

There are 3 scheduled tasks:

- every 1 minute the script `start_auto_restart_check_cron` is run to ensure that the `auto_restart` script is running.
- every 1 minute the script `start_procmap_check_cron` is run to ensure that `procmap` is running.
- every 5 minutes `start_build_logdir_links` runs to create symbolic links in the log directories to point to log files for yesterday and today. The log files are stored in directories named for the date, i.e. `yyyymmdd`. The links are a convenient way to easily find the log files for today and yesterday.

The process mapper, ‘procmap’.

The process mapper, `procmap`, lies at the heart of the auto-restart capabilities of the TITAN system. `procmap` keeps a table of the current status of all processes running on the system, except for itself. Each running processes registers with `procmap` at regular intervals, usually once per minute. This is called the ‘heart-beat’ interval. The process status table is read from `procmap` by the `auto_restart` script and compared against the list of expected processes in the `proc_list`. If a process is missing or has not registered its heartbeat recently, it is killed (in case it is hung) and then restarted.

procmap may be queried by the application 'print_procmap' which prints out the current table of processes, along with status information.

The data mapper, 'DataMapper'.

The DataMapper performs a task similar to procmap, except for data sets instead of processes. The DataMapper keeps a table of all data sets on the system, along with such information as the last time data was added to the data set, how many files exist in the data set and how much disk space it occupies.

Each time an application writes data to disk it also registers that activity with the DataMapper. That allows the DataMapper to keep an up-to-date status table. The DataMapper table may be queried by the application 'PrintDataMap', which then prints the table information.

The auto_restart scripts.

There are 3 scripts which are part of the auto_restart system:

auto_restart: this is the most important script. It is responsible for contacting procmap at regular intervals, say once per minute, and checking the table of processes which are running against the proc_list. Any processes which are missing or late in registering are killed with the kill script or kill_inst mechanism (in case they are hung) and then restarted with the start script.

procmap_list_start: this script is used at system startup to go through the proc_list and start all processes by calling the start scripts.

procmap_list_kill: this script is called at system shutdown to go through the proc_list and kill all processes by calling the kill script or the kill_inst mechanism.

Log files.

There are 2 sets of log files. The error logs reside in ~/projDir/logs/errors and the restart logs in ~/projDir/logs/restart. The logs files reside in sub-directories named using the date. As an example, the error logs for September 25 2005 will reside in ~/projDir/logs/errors/20050925. For convenience a link is provided to the logs from yesterday and today. The link is updated every 5 minutes.

The error logs contain all error messages generated by the processes themselves. The log files are named after the process name and instance. For example, the PrecipAccum application running as instance 24hr will create a log file called PrecipAccum.24hr.log.

The restart logs are written by the auto_restart script and show all restart activity.

The log files are managed by piping the stderr and stdout output from processes through a specially-written filter called LogFilter. This application reads data from standard input

and writes it to a daily file.

The Janitor.

The `Janitor` application is used by the real-time system to keep the disk from filling up. A full disk is fatal for any system running in real-time with new data arriving, since when the disk is full no new data will be written and the system will fail.

The `Janitor` has three major functions: (a) to delete files which are older than specified age, (b) to delete empty directories and (c) to compress files which are older than a specified age.

WARNING: the Janitor is a potentially **DESTRUCTIVE** program which will faithfully delete whatever you tell it to delete. So be careful to set it up correctly. In the parameter file you can set report mode on so that the Janitor will leave small text files at the nodes visited to aid in debugging.

The `Janitor` operates by traversing the data directory tree, starting at an entry point referred to as the '`top-dir`'. Normally it starts at `$DATA_DIR`. By default the `Janitor` does nothing other than traverse the directory tree looking for parameter files named `_Janitor`.

When it finds an `_Janitor` parameter file, it reads in that file and uses it to override the current settings. The overridden settings only apply from that point DOWN in the tree, and until another `_Janitor` file is found. As the `Janitor` pops back up the tree it reverts to using the parameters which were in effect at the upper level, before descending to lower levels.

To set up the `Janitor`, normally a `_Janitor` file is placed in `$DATA_DIR`, to specify the starting behavior as it traverses the tree. At the top level it is normally set up to do nothing. Then, `_Janitor` files are placed at positions in the data tree designed to control the behavior from that point down. If you need all files below a given point to be deleted after 5 days, set the parameters at that level to indicate that preference. Similarly for compression.

There are a number of file types which the `_Janitor` will not delete. These include files beginning with underscore '_'. That is the reason that the name of any parameter file in the data area always starts with an underscore.

The Janitor can be set up to save data in so-called 'event lists'. If you want to age off all data except that for certain events, put this information in the event list at the top level.

The early version of the Janitor only specified file ages in seconds. This became cumbersome, but was retained for backward compatibility reasons. Some parameters were added to allow you to specify the ages in days, instead of seconds. These are as follows:

```
////////////// file_ages_in_days //////////////////////
//
// Option to specify file ages in days, instead of secs.
// If TRUE, 'MaxNoModDays' and 'MaxNoAccessDays' are used. If FALSE,
//   'MaxModificationAgeBeforeDelete' and
// 'MaxAccessAgeBeforeCompress' are used.
```

```
// Type: boolean
//

file_ages_in_days = FALSE;

////////////// MaxNoModDays ////////////////////////////
//
// Max file age before deletion - days. Used if 'file_ages_in_days'
//   is TRUE.
// If delete_files is TRUE, a file will be deleted if it has not been
//   modified in this amount of time.
// Type: float
//

MaxNoModDays = 30;

////////////// MaxNoAccessDays /////////////////////////
//
// Max quiescent age before compression - days. Used if
//   'file_ages_in_days' is TRUE.
// If compress is TRUE, a file will be compressed if it has not been
//   accessed within this time.
// Type: float
//

MaxNoAccessDays = 1;
```

There is another parameter which can cause confusion, named date_format. By default it is TRUE.

```
////////////// date_format ////////////////////////////
//
// Option to check for RAP date-time naming convention.
// If set, the files must follow the RAP file naming
// convention (which means that the filename is based
// on the date that the data in the file pertains to).
// Type: boolean
//

date_format = TRUE;
```

If date_format is left `TRUE`, The `Janitor` will only delete files which conform to certain naming conventions related to the date and time. This works fine for most files within the TITAN system. However, it often will not work with raw files provided from other sources. Therefore, to be sure to delete all file types, set this to `FALSE`.

You can tell the Janitor to avoid certain parts of the tree altogether. This saves CPU and ensures you will not delete anything in that area.

```
////////////// recurse ////////////////////////////////
//
// Recurse to lower directories.
// Set to false to leave directories below the current directory alone.
// Type: boolean
//

recurse = FALSE;
```

You can also tell the Janitor to avoid processing a single directory while progressing to lower directories:

```
////////////// process ////////////////////////////////
//
// Process files in this directory.
// Set to false to leave the current directory alone.
// However, subdirs are processed unless the recurse
// parameter is set to false.
// Type: boolean
//

process = FALSE;
```

This can get pretty confusing, however, so normally `recurse` and `process` are both set to `FALSE` together to protect a directory.

The Scout.

The Scout is a program with properties similar to the Janitor, except that instead of deleting or compressing files it scans the directories for information about the data sets and registers that information with the DataMapper.

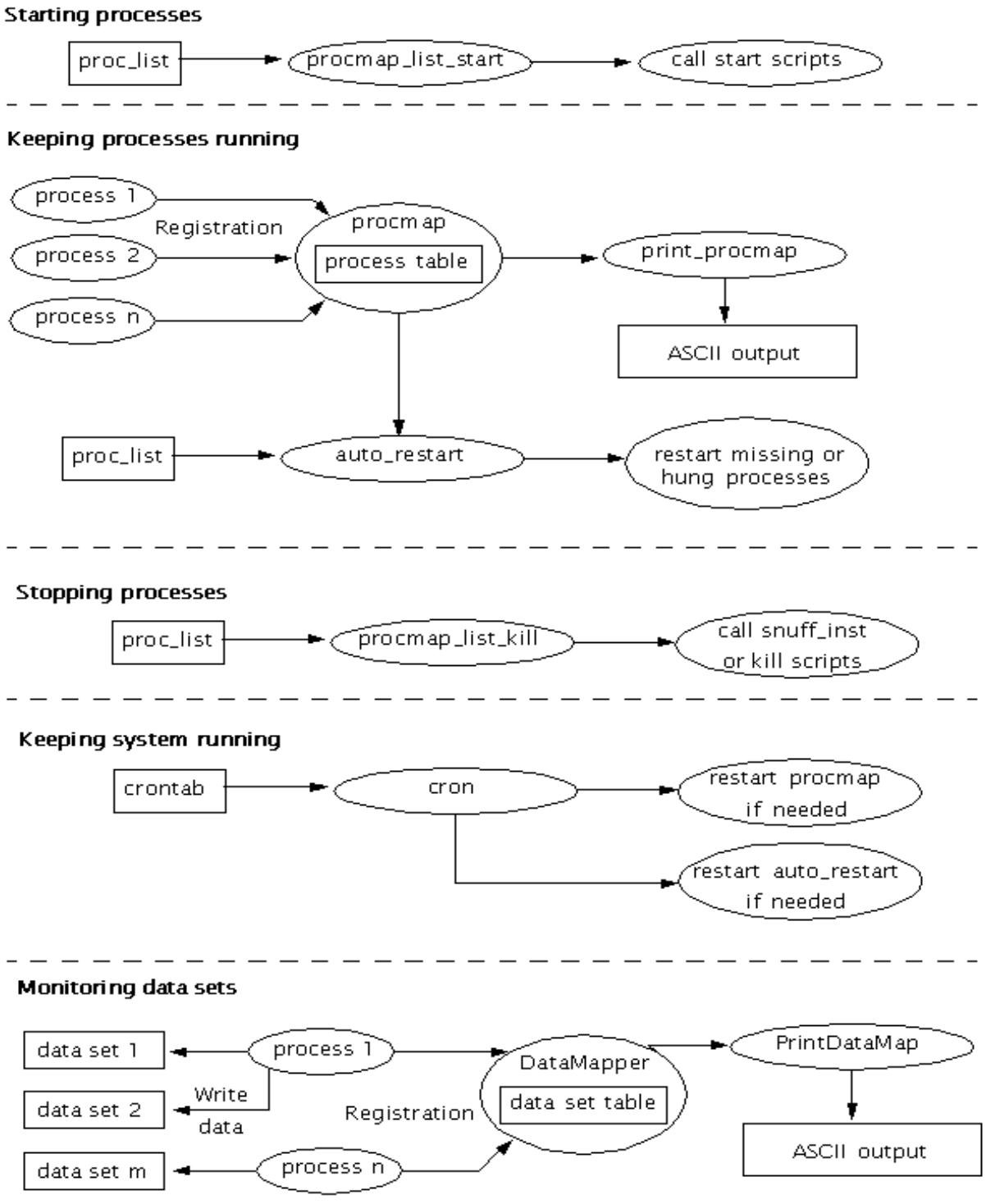
The status information gathered on a data set by the Scout is:

- start date
- end date
- number of files
- number of bytes

The Scout is useful because it helps summarize the status of the data sets. However, it is not essential to the operation of the real-time system.

Control flow in the real-time system

It is useful to graphically visualize the flow of control and information in the real-time system. The figure below shows this control flow.



Starting the real-time system

To start the system on a host, type the command:

```
start_all
```

This script performs the following steps:

- Starts the process mapper procmap.
- Starts all of the processes listed in the process list file:

  ```
  ~/projDir/control/proc_list
  ```

- 3. Starts the auto_restart script.
- 4. Installs the cron table:

  ```
  ~/projDir/control/crontab
  ```

To check that the system started correctly, type the command:

```
pcheck
```

This script checks that all of the required processes have been successfully started. You should get the message:

```
0 processes down
```

If any processes are down, check that the start scripts and that you can successfully start them by hand. Frequently problems with this step are related to typos which are difficult to spot.

Stopping the real-time system

To stop the system on a host, type the command:

```
stop_all
```

This script performs the following steps:

- Removes the `crontab`.
- Stops the `auto_restart` script.
- Stops `procmap`.
- Stops all of the other TITAN processes.
- Removes any shared memory segments.

Starting and stopping individual processes

There are two scripts, `snuff` and `snuff_inst`, which are useful for stopping individual processes.

To kill all processes with a specified name, run the command:

```
snuff process_name
```

To kill all processes with a specified name and instance, run the command:

```
snuff_inst process_name instance
```

To start a process, just call the relevant start script. For processes which appear in the proc_list, the `auto_restart` script will restart the process anyway.

Quick check on the real-time system

To check that all processes are running, type the command:

```
pcheck
```

This is an alias for:

```
procmap_list_check -proc_list ~/projDir/control/proc_list
```

This will report on any processes which are down. For example, if the DataMapper is down you would see:

```
1 process(es) down
   DataMapper primary missing
```

If all processes are running you will get the message:

```
0 processes down
```

If any processes are down, check that the start scripts and that you can successfully start them by hand. Frequently problems with this step are related to typos which are difficult to spot.

Detailed check: print out all processes

To print a table of all processes running on a host, type the following command:

```
ppm
```

`ppm` is an alias for:

```
print_procmap -hb -up -status
```

You can check the processes on a remote host:

```
ppm -host hostname
```

To see the print repeated every 5 seconds, type:

```
ppm -c 5
```

ppm will produce a listing like the following:

```
PROCS REGISTERED - localhost - Sat Sep 24 15:39:54 2005
Uptime: 25.2 d

Name         Instance  Host    User   Pid    Heartbeat  Uptime   Status
====         ========  ====    ====   ===    =========  ======   ======
DataMapper   primary   helene  titan  11653  0:0:50     2:23:51  Listening, port: 5434
DsServerMgr  primary   helene  titan  11475  0:0:53     2:23:54  Listening, port: 5435
Dsr2Vol      ops       helene  titan  11941  0:0:44     2:23:45  In FMQ_read_blocking()
EsdAcIngest  sim       helene  titan  11868  0:0:46     2:23:47  Reading data
Janitor      primary   helene  titan  11490  0:0:53     2:23:54  Sleeping between passes
Mdv2Vil      ops       helene  titan  12160  0:0:34     2:23:35  LdataInfo::readBlocking
PrecipAccum  1hr       helene  titan  12073  0:0:38     2:23:39  LdataInfo::readBlocking
PrecipAccum  24hr      helene  titan  12117  0:0:36     2:23:37  LdataInfo::readBlocking
PrecipAccum  single    helene  titan  12027  0:0:40     2:23:41  LdataInfo::readBlocking
RadMon       ops       helene  titan  12349  0:0:28     2:23:29  In FMQ_read_blocking()
Rview        ops       helene  titan  12286  0:0:30     2:23:31  In event loop (OK)
Scout        primary   helene  titan  11505  0:0:52     2:23:53  Sleeping between runs
Test2Dsr     sim       helene  titan  11751  0:0:48     2:23:49  Creating next beam
TimeHist     ops       helene  titan  12290  0:0:30     2:23:31  In event loop (OK)
Titan        ops       helene  titan  11984  0:0:42     2:23:43  DsMdvxTimes::getLatest
Tstorms2Spdb ops       helene  titan  12203  0:0:32     2:23:33  LdataInfo::readBlocking
```

The columns in the above list have the following meanings:

Name:

The process name.

Instance:

The process instance. There may be more that one instance of a program running. Therefore, the instance is required for a unique reference to each instance.

Host:

The host on which the process is running.

User:

The user who started the process.

Pid:

The process ID.

Heartbeat:

The time since the latest heartbeat, in seconds. Each program registers with the process mapper (procmap) at regular intervals, normally every minute. The heartbeat time gives the time since the last heartbeat.

If you run 'ppm -maxint', you will see the maximum heartbeat interval, which is generally twice the normal heartbeat interval. So for most processes this will be 120 secs. If the

process fails to heartbeat within this interval, it will be killed and restarted by the auto_restart script.

Uptime:

The time, in secs, since the process started.

Status:

The status message sent when the process last registered.

Checking the data sets

To print a table of all data sets available on a host, type the following command:

```
pdm
```

Or, for data sets on a different host:

```
pdm -host hostname
```

To see the print repeated every 5 seconds, type:

```
pdm -c 5
```

`pdm` is an alias for:

```
PrintDataMap -all -relt -lreg
```

`pdm` will produce a listing similar to the following:

```
========== Data on host 'localhost' at time 2005/09/24 22:13:57 ==========

DataType               Dir HostName    Latest  Last reg Start date   End date nFiles nBytes
========               === ========    ======  ======== ==========   ======== ====== ======
     mdv     mdv/precip/1hr   helene -00:02:13 -00:02:07 2005/09/24 2005/09/24     12   9.4M
     mdv    mdv/precip/24hr   helene -00:02:13 -00:02:07 2005/09/24 2005/09/24     12   9.4M
     mdv  mdv/precip/single   helene -00:02:13 -00:02:08 2005/09/24 2005/09/24     12   1.2M
     mdv     mdv/radar/cart   helene -00:02:13 -00:02:09 2005/09/24 2005/09/24     12    30M
     mdv            mdv/vil   helene -00:02:13 -00:02:08 2005/09/24 2005/09/24     12   772K
    spdb       spdb/ac_posn   helene -00:00:01 -00:00:01 2005/09/24 2005/09/24      2   238K
    spdb spdb/ascii_ac_posn   helene -00:00:01 -00:00:01 2005/09/24 2005/09/24      2   201K
    spdb       spdb/tstorms   helene -00:02:13 -00:02:07 2005/09/24 2005/09/24      2    41K
   titan       titan/storms   helene -00:02:13 -00:02:08 2005/09/24 2005/09/24      4   424K
                                                                                ====== ======
                                                                                    70    52M
```

The columns in the above listing have the following meanings:

DataType: category of data, for example:

`raw`: data in native input format

`mdv`: gridded format

`spdb`: symbolic product format (non-gridded)

`titan`: TITAN storm and track files

Dir:

The directory for the data. This is relative to `$DATA_DIR`, which is normally `~/projDir/data`.

HostName:

The hostname on which the data is stored.

Latest:

The time of the latest data on the disk. This is relative to 'now'.

Last reg:

The time at which the data set was last registered with `DataMapper`. This is relative to 'now'. The process which writes the data to disk is responsible for registering with `DataMapper`.

The last registration time gives you an idea about whether the data is coming in on time.

Start date:

The start date of the data set. This is reported by the `Scout`.

End date:

The end date of the data set. This is reported by the `Scout`.

nFiles:

The number of files in the data set. This is reported by the `Scout`.

nBytes:

The number of bytes in the data set. This is reported by the `Scout`.

Changing the process list on the fly

You can change the process list without having to restart the entire system.

If you add a process/instance to the `proc_list`, the `auto_restart` script will try to start that process then next time it checks the list.

If you remove a process/instance from the `proc_list`, you will need to call `snuff_inst` to kill that process.

Changing the cron table on the fly

You can change the cron table without having to restart the entire system.

If you make a change to the crontab file, you can activate that table by running the command:

```
start_cron
```

To see the active cron table, run the command:

```
crontab -l
```

To remove the current cron table, run the command:

```
kill_cron
```

Running TITAN applications in ARCHIVE mode

In REALTIME mode, the various TITAN applications watch for incoming data and perform actions when new data arrives. Often this action involves writing out results based on running analyses on the incoming data. This forms an alternating chain of data and applications in which the output from one application forms the input to the next. It is therefore necessary to have all of the applications running at the same time.

In ARCHIVE mode, on the other hand, it is not necessary for all of the applications to run at the same time. You can repeatedly run a program in ARCHIVE mode on data that was produced once by the upstream application.

Many TITAN applications have such an ARCHIVE mode capability.