

File Mover

The File Mover utility, previously named "Data Mover," will be used in Pre-Production and Production to copy and move files as follows:

- Copy files from DDE Production to ECE Pre-Production for parallel testing.
- In ECE Production, *temporarily* move files as LOBs work on moving inputs that currently point to DDE to instead point to ECE.

File Mover Components

The File Mover utility comprises the following components:

Component	Description
abfsmon	A daemon process that monitors filesystem events and uses user-defined rules to determine when a set of files is ready to be copied. When a set of files is ready to be copied, abfsmon generates a <i>copy batch file</i> for that set. The copy batch file lists the files and other details needed by downstream file-copy jobs.
file_mover.plan	Processes a given copy batch file, copying data files and optionally copying or creating trigger files.
Dispatcher job	For each generated copy batch file, you must create a corresponding Control Center dispatcher job definition. These dispatcher jobs run the file_mover.plan whenever a file appears matching the naming pattern specified in the given copy batch file.

abfsmon Syntax and Usage

This section describes **abfsmon** syntax, and provides usage instructions, examples, and notes.

abfsmon Syntax

```
abfsmon [-h] [--error-file <filename>] [--log-file <filename>]
        [--list-period <seconds>] [--update-timeout <seconds>]
        [--wait-for-unlock] [--daemon]
```

```
[--logging-level {CRITICAL | ERROR | WARNING | INFO | DEBUG}]  
<list_directory> <config_file>
```

Positional arguments	
<code><list_directory></code>	Directory in which copy batch files and state information is written.
<code><config_file></code>	Name of a configuration file that contains a list of watched directories and matching patterns. For more information, see abfsmon .config File .

Optional arguments	
<code>--error-file <filename></code>	Error messages filename.
<code>--log-file <filename></code>	Log filename.
<code>--list-period <seconds></code>	Period in seconds to use for grouping copy batches.
<code>--update-timeout <seconds></code>	Time in seconds to wait without modification before designating a file as ready for copying.
<code>--wait-for-unlock</code>	Check for the release of <code>.lock</code> files before copying.
<code>--daemon</code>	Run as a daemon.
<code>--logging-level {CRITICAL ERROR WARNING INFO DEBUG}</code>	Level of detail to capture in the log file.
<code>--automatic-restart</code>	Automatically restart <code>abfsmon</code> after an unexpected failure.

abfsmon .config File

The `abfsmon` command requires an input `.config` file in which you specify the following information:

- The directories to watch for newly created files.
- The filename patterns to match when generating the list of files to copy.
- How to determine when a file is no longer being written and is ready to be copied.
- Whether the file is a regular file or an mfile.
- Whether a trigger file should be created alongside the copied file.

The specific layout of the `.config` file varies slightly depending on the kind of processing you want to perform. In general though, each line of a given `.config` file will contain `|`-delimited fields carrying the required information. For more information, see the [Examples](#) later in this chapter.

Determining When a File is Ready to be Copied

Every file to be copied will match a filename and directory pattern in the `.config` file. There are two different ways that you can configure `abfsmon` to determine when a file is ready to be copied:

1. The most reliable way is to wait for a small [trigger file](#) to be created after the data file has been

written.

- Alternatively, if the upstream process does not create a trigger file, you can use the `update_timeout` option to tell `abfsmon` to list a file for copying once it has received no updates for some amount of time.

More About Trigger Files

In cases where the upstream process creates trigger files, entries in the `.config` file will take the following form:

```
<trigger_file_name_pattern>|<data_file_name_template>|<watch_directory>|[file | mfile]
```

<code><trigger_file_name_pattern></code>	A regular expression pattern for the trigger filename. The pattern language is Perl-like and is compiled using the Python <code>re</code> module. The main difference between standard Perl and the syntax used here is that the <code> </code> operator character is not allowed in the pattern because this is used as the <code>.config</code> file field delimiter. This pattern should generally include capturing groups to be used in the <code><data_file_name_template></code> .
<code><data_file_name_template></code>	A template that will be expanded based on the match object created when <code><trigger_file_name_pattern></code> finds a match. Capturing groups appearing in <code><trigger_file_name_pattern></code> are referenced using typical regular expression backreference notation (for example, <code>\1</code> , <code>\2</code>).
<code><watch_directory></code>	The full path to the directory in which the files are being created.
<code>[file mfile]</code>	Specifies whether the file is a standard file or a multifile.

Examples

Example 1

In this first example, we watch for a file named like `YYYYMMDD_data_filename.dat`, which will be uploaded to the directory `/data/incoming`. We know that, as soon as the upload is complete, the job sending this file will send a 0 byte file correspondingly named `YYYYMMDD_data_filename.dat.trg`.

One way of specifying this in the `.config` file would be as follows:

```
(.*_data_filename\.dat)\.trg$|\1|/data/incoming|file
```

In the `<trigger_file_name_pattern>` field, a capturing group (indicated by the parenthesis), surrounds the part of the name that will match the data file. The `<data_file_name_template>` simply uses a backreference to that capturing group to name the corresponding data file.

Example 2

As a second example, we watch for a data file named like `dependent_data_file1_YYYYMMDD.dat`. This file will be created by a complicated graph job that writes many files. We may not know exactly when the data file that we are interested in will be done being written, but we do know that the complex job will write a small file named like `complex_job_YYYYMMDD.complete` when it has finished. We can wait until that `.complete` file is created to start copying.

To do this, one approach would be to specify the following in the `.config` file:

```
complex_job_(.*).complete$|dependent_data_file1_\1.dat|/data/complex_job_workdir|mfile
```

With this approach, most of the trigger filename is unrelated to the data filename, but we still need to make certain that we match the date part of the name. Otherwise, we could be copying a data file from the wrong day. We surround the date part of `<trigger_file_name_pattern>` with a capturing group and reference back to that group with the backreference notation `\1` in the `<data_file_name_template>`.

An alternative approach for this example is to use the `update_timeout` parameter. In this approach, when a trigger file is unavailable, you can use `update_timeout` to tell `abfsmon` to wait until a data file has not received any new data for some specified amount of time. The `update_timeout` parameter controls how many seconds `abfsmon` should wait without seeing updates.

To use this approach, the `.config` entry would be as follows:

```
|<data_file_name_pattern>|<watch_directory>|[file | mfile]
```

This is the same syntax as for the trigger entry, except that now the `<trigger_file_name_pattern>` field is left empty, and the second field should be a regular expression matching the data filename instead of a template used for resolving it.

Added Triggers

Sometimes `abfsmon` may be watching a file that does not have a corresponding trigger file, but the downstream process reading the data file that the File Mover copies expects to see a trigger file. In this situation, `abfsmon` can create a trigger file alongside the copied data file after it copies the data file to its target.

To enable this behavior, you must append a special field to the regular `.config` entry for the watched file. In this case, the entry would take the following form:

```
|<data_file_name_pattern>|<watch_directory>|<file_or_mfile>|<add_trigger_extension>
```

where `<add_trigger_extension>` is a file extension to append to the name of the data file.

When a file is configured in the manner, the downstream copying component will create a file alongside the copy target, using a name like the following:

```
data_file_name.dat.trg
```

where 'trg' is the value found in the `<add_trigger_extension>` field.

Usage Notes

Note the following:

- Name patterns are used to match the **basename** of a file and should not include the directory part of any paths.
- Name patterns must be specific enough to not include other files that might be created. For example, when describing a pattern, is a good idea to include a '\$' at the end of the pattern to match the end of a string. Without such a delimiter, it is easy to inadvertently match lock files, control files, and suchlike that should not be copied, but have similar names to data files; for example, `filename.dat.mfctl`.
- Using the watched file approach to determining whether a multifile is complete can be problematic and is not recommended. This is because a multifile is really a group of files, and in certain data processing scenarios it is relatively easy for some data partition files to finish writing long before other data partition files receive any data. This makes the "safe-to-copy" `<update_timeout>` dependent on details of a particular graph and the environment(s) that it runs in. In the most problematic scenarios (which are not difficult to construct), the required `<update_timeout>` becomes proportional to a data partition's processing time or data volume — for example, the difference in data volume for partition 0 and partition 1.

Running abfsmon

Notes on Running:

- It is expected that only one instance of `abfsmon` will run for a given `<list_directory>`. `abfsmon` self-enforces this when it starts up by creating a pid lock file in the `<list_directory>`. If such a file already exists, then `abfsmon` concludes that another instance of `abfsmon` is already using the `<list_directory>`.
- If the `.pid` lock file is somehow left in place due to a failure, but `abfsmon` is no longer running, then it will be necessary to delete the `.pid` file before `abfsmon` will start. Typically, if a failure does happen, the `.pid` file will be removed gracefully.
- When running as a daemon, `abfsmon` can be cleanly shutdown by sending it a SIGTERM signal; for

example:

```
kill `cat listdir/abfsmon.pid`
```

If it is not running as a daemon, then it should respond to a Control-C keyboard interrupt.

Event Processing

abfsmon processes events in the following order:

1. Process file any queued CREATE events:
 - Any trigger file CREATE events are immediately added to the current copy batch.
 - Any watched file CREATE events are added to a list of watched files.
2. Process watched files:
 - If abfsmon time greater than *<update_timeout>* has elapsed for any watched file, the mtime for that file is updated.
 - For any files that have an *mtime* older than *update_timeout*:
 - If *wait_for_unlock* has been set and a corresponding .lock file exists, then do nothing.
 - Otherwise, add the watched file to the current copy batch
3. Release copy batch:
 - If *<list_period>* has been exceeded, create a copy batch file listing files to be copied
4. Repeat.

The abfsmon copy batch File

The copy batch file is how abfsmon notifies downstream components that files are ready to be copied.

While a copy batch is still in use by abfsmon — that is, while copy batch is still having files added to it — copy entries will be stored in a file in *<list_directory>* using the following naming convention:

```
filelist-YYYYMMDD-HH24MISS
```

where the timestamp corresponds to the time that the batch was created.

Once the batch file is no longer in use by abfsmon and entries are ready for copying by the downstream components, the copy batch file is renamed to have a .lst extension:

```
filelist-YYYYMMDD-HH24MISS.lst
```

Entries in the copy batch file are formatted as follows:

<directory_name>|<data_file_name>|<copy_method>|<trigger_file_name>

where <copy_method> will be one of the following:

T	Triggered data files.
N	Watched data files.
A	Watched data files that require a new trigger file be created upon copying.

In the cases of the **T** or **A** copy methods, <trigger_file_name> will name the trigger file that is to be copied or created, respectively.

abfsmon Logging

Log output generated by abfsmon is sent two files that configured by the command line arguments:

- --error-file <error_file>
- --log-file <log_file>

Any error or warning messages will go to the <error_file>. The <log_file> will receive the same error messages as the <error_file>, and can be configured to include more detailed messages using the logging-level command-line argument:

--logging-level [CRITICAL|ERROR|WARNING|INFO|DEBUG]

The default logging level is INFO, which is generally enough information to capture the most interesting events.

Because these log files can grow large, the abfsmon logger will automatically rotate them at midnight, renaming log_file.log to log_file.log.1, log_file.log.1 to log_file.log.2, and so forth.

Failure Recovery

What happens if abfsmon unexpectedly fails?

If abfsmon unexpectedly fails, it will attempt to save its current state in .state files that are located in <list_directory>. During such controlled, albeit unexpected, shutdowns, abfsmon performs the following steps:

1. Writes a snapshot for each <watch_directory> to disk.
2. Writes a list of any yet unhandled CREATE events to disk as *pending events*.
3. Writes a list of any watched files to disk.
4. Releases the current copy batch for copy.

5. Releases the pid file lock.

What happens when abfsmon restarts after failure?

When `abfsmon` starts, it looks in `<list_directory>` for `.state` files from any previous runs. If it finds any such files, it then performs the following recovery actions:

1. Begins listening for `CREATE` events in each `<watch_directory>`.
2. Handles watched files' previous states:
 - Reads previous state for watched files.
 - Adds any previously watched files to the current watch list.
 - Immediately processes watched files as would be done in the normal event processing loop.
 - Removes the associated `.state` files.
3. Processes `<watch_directory>` snapshots:
 - If a directory's current `mtime` has not changed since the last run, assume any creation events associated with this directory are already accounted for in the *pending events* set.
 - Any files currently listed in a `<watch_directory>` that do not appear in the snapshot are added to the *pending events* set.
4. Adds previously saved *pending events* to the *pending events* set.
5. Adds any `CREATE` events that occurred during the current run of `abfsmon` to the *pending events* set.
6. Processes *pending events*:
 - A deduped set of *pending events* is handled as would normally be done for `CREATE` events during ordinary event processing.
 - Removes the associated `.state` file.

What is automatic restart?

You can instruct `abfsmon` to automatically relaunch with the same configuration after an unexpected failure. For this to happen, the failure needs to be handled in such a way that a controlled shutdown was possible.

So, for example, a controlled shutdown would not happen if `abfsmon` is sent `SIGKILL`, or if a failure occurs that is severe enough to break `abfsmon`'s more general exception handling machinery.

The restart feature has been shown to help in mitigating recoverable failures caused by unexpected or unhandled temporary environmental conditions. A hypothetical (though not necessarily existing) example of such a situation would be an unhandled failure in listing a multiframe due to a short-lived and temporary networking issue. Such a failure would likely lead `abfsmon` into a controlled, but unexpected, shutdown. However, upon automatic restart, it would likely continue without problems.

Control Center Dispatcher Jobs

You must create a Control Center dispatcher job that will be triggered whenever a new `filelist-YYYYMMDD-HH24MISS.lst` file appears in the `<list_directory>`. The dispatcher job will run a corresponding [file_mover.plan](#) with the appropriate input parameters.

For more information about dispatcher jobs, see the Control Center documentation.

file_mover.plan

The `file_mover.plan` performs the actual file copying work, takes the following input parameters:

FILE_LIST_NAME	The path to a <code>filelist-YYYYMMDD-HH24MISS.lst</code> copy batch file created by <code>abfsmon</code> .
DESTINATION_HOST	The name of the host that files will be copied to.
SOURCE_HOST	The name of the host that files will be copied from.
SRC_TO_DST_MAP	The path to a <code>.map</code> file used to configure <code>'file_mover.plan'</code> .

Of these, `FILE_LIST_NAME` will vary, and should be provided by the Control Center dispatcher job. The rest will generally be fixed.

.map File Format

The `.map` file that is used by `file_mover.plan` takes the following form:

```
<source_directory>|[file | mfile]|<target_directory>|[file | mfile]
```

The `<directory_name>` listed in a copy batch file is matched to a `<source_directory>` in the `.map` file in order to determine the `<target_directory>` for copying.

The `[file | mfile]` fields exist for historical reasons, but do not change the behavior of the plan. It is expected that the source and target are either both `file` or both `mfile`. For multfiles, the source and target must be of the same depth.