

---

# Disk and Tape Archiving for SPOL

Gary Granger

National Center for Atmospheric Research

Atmospheric Technology Division

\$Id: spol-archiving-notes.xml,v 1.6 2004/05/19 07:49:39 granger Exp \$

This article contains notes about the disk and tape archiving components in SPOL/RDA.

## Table of Contents

1. Latest Changes .....	1
2. Disk Archiving .....	1
3. Tape Archiving Process: spol_tar .....	2
4. The TarQueue.py module .....	2
4.1. Requirements .....	2
4.2. Implementation .....	3
4.3. Input and Output Catalogs .....	5
4.4. Status Information .....	5
4.5. XML-RPC Methods .....	6
4.6. Tape Format .....	7
4.7. Verifying Tapes .....	7
4.8. Batch Control with xquery .....	7
5. GUI Status and Control .....	8
6. File Catalogs .....	9

## 1. Latest Changes

This section is for listing the latest changes to the SPOL archiving software and this document, in reverse chronological order, so the more recent changes appear first.

**Added Section 4.8.** There is now a section about how to control `TarQueue` from the command line or in a batch mode using the **xquery** XML-RPC command-line client.

**New unloading state.** `TarQueue` now has an unloading state so that a pending **tar** process can finish before the tape is taken offline. See `TarQueue States`.

**Add tape status checks.** The tape drive status is now checked more thoroughly with `ioctl()` calls and by explicitly loading and unloading the tapes. See Section 4.2.1.

**Circular message buffer in `TarQueue` status.** A rolling message buffer has been added to the `TarQueue` status structure so the last several status messages can always be displayed in the GUI controller window.

**Added the Latest Changes section.** This document needed a place to record updates and changes so that new information in each revision can be found and reviewed easily.

## 2. Disk Archiving

The disk archiver receives radar beam data from multiple streams and writes the data blocks into files on disk. Files are limited to a certain size, causing a new file to be created when the existing one reaches the limit. The disk files must ultimately be archived to tape, so when the disk archiver creates a file it also inserts the file into multiple catalogs. (See Section 6 for more information on file catalogs.) One catalog contains a complete list of all of the files generated by the disk archiver:

```
/data/archive_spol/spol_archiver.catalog
```

The second catalog is the input queue to the tape archiving process:

```
/data/archive_spol/spol_archiver-tar.catalog
```

When a file entry is first inserted into the queue, the disk archiver tags it with a state of open. This informs the tape archiver that though the file is in its queue, tape archiving should be deferred until the disk archiver has finished writing to it. In the usual case, the disk archiver finishes writing a file and changes its state to closed, indicating to the tape archiver that the file is ready to be written to tape. This handshaking is necessary to account for potential disk archiver crashes or interruptions before a file is finished. In that case, there still needs to be a record that the incomplete file was created so that it can be written to tape. Thus the file entry also contains an expires field, a timestamp denoting when the tape archiver can safely assume the file is no longer being written, even if its state still set to open. The disk archiver updates both of its catalogs simultaneously, but the tape archiver will only modify the catalog serving as its input queue.

The SPOL archiver is actually an instance of a CORBA `ArchiverService`, whose interface is defined in this IDL file within the `raddx` source tree: `raddx/radd/archiver/archiver.idl`. The same implementation of this service (in C++) is shared by both SPOL and ELDORA.

## 3. Tape Archiving Process: `spol_tar`

The tape writing is run by the `spol_tar` python script, located here in the `raddx` source tree:

```
raddx/spol/archiver/spol_tar
```

The script gets installed into `/opt/spol/bin`. It depends on a few python modules, also in the `spol/archiver` directory, which should all install into `/opt/spol/lib/python`: `TarQueue.py` and `FileEntry.py`. The crux of the work happens in `TarQueue.py`.

## 4. The `TarQueue.py` module

## 4.1. Requirements

There are a few critical requirements. Most important: **tar** files to tape and record that they have been successfully written to tape. The recording is necessary so that a file can be scrubbed from disk only after it is known to have been archived. Also important: detect when the tape is full, and related to that but less important, keep track of how much has been written to the tape so far.

A few alternative implementations were considered. One option: **tar** could be run as a child process opening and writing directly to the tape device, and the exit status could be used to determine success or failure. However, this means other processes could potentially use the tape device between calls to **tar** and confuse the state of the tape device. This is not all that unlikely since a rogue or remnant instance of the tape archiving program would cause competition for the tape device. The Zebra archiver actually opens a pipe on the output from **tar**, then forwards the data from the pipe to the tape device. This allows the Zebra archiver to keep the tape device open and prevent any contention. Also, it allows detection of end-of-tape (EOT) since the archiver can notice short writes, and the archiver can count the bytes redirected from the pipe. However, the redirection could incur significant overhead which could prevent the throughput needed for tape streaming.

## 4.2. Implementation

The current implementation uses file descriptor sharing to pass the tape device file descriptor to the child **tar** process. The program opens the tape device and duplicates that file descriptor on `stdout` with `dup2()`. The child process inherits `stdout`, so **tar** writes its output directly to the tape device. The program can keep the tape device open as long as its running. The parent program must be careful, though, not to write any other output to `stdout`. Unfortunately, with this scheme there is no easy way to catch the `stderr` output from **tar** separately. Instead, the parent program's log output is written to `stderr` where it will merge with the tar output, and `stderr` can optionally be redirected to a log file.

Originally, the design intended to keep the tape device open only while a tape was loaded and files were being written to it. However, that proved problematic for the `SimpleXMLRPCServer` Python class being used to service XML-RPC status requests. The symptoms were that any attempt to re-open the standard output file descriptor (1), such as with the `dup2()` or `fdopen()` calls, would subsequently cause exceptions in the XML-RPC handler code with the error socket operation on non-socket. To work around this problem, the tape device is opened only once, `dup2()` is called only once, and both file descriptors are kept open the entire time the `TarQueue` process is running.

### 4.2.1. Tape Control and Status

One consequence of not re-opening the tape device for each new tape is that the tape drive must be controlled more directly with `ioctl()` calls. When re-opening the drive device for each new tape, the drive would automatically report that the drive was online and a tape had been loaded. However, when the drive device is kept open during tape reloads, the drive must be

told explicitly to load the new tape. Otherwise, the drive status will continue to report that the drive is offline and open (no tape loaded) long after a tape has been inserted. After telling the drive to load the tape, `TarQueue` queries the drive status and verifies that the drive is online, a tape is loaded, and the tape is not write-protected.

### Note

Actually, if the tape is write-protected, the status query fails with the error message `read-only filesystem because the tape device was opened for writing`. Hopefully the implication of that message is obvious enough. According to the documentation for the Linux SCSI tape module `st.o`, not all drives check that the tape is not write-protected just because the device was opened for writing.

Despite the check on the drive status after loading a tape, for some reason it can still happen that writes to the tape fail just after the drive reports that it is online. Given this possibility, a failed `tar` no longer forces an unload of the tape. Instead, the failed `tar` is reported in the status messages and retried on the next attempt to process the queue. It is up to the operator to notice repeated failures and decide whether to manually unload the current tape and load a new one.

The decision to retry `tar` attempts is more palatable given that the `ioctl()` status query now allows detection of an actual end-of-tape condition (EOT). As long as EOT detection is reliable, it should be possible to differentiate between `tar` attempts which fail because the tape is full and other transient failures. The `TarQueue` status messages will report an EOT condition with a `Tape is full` message and automatically unload the tape.

### 4.2.2. Low-level Tape Drive Interface

The `TarQueue` class uses the Python module `mtio.py` as a simple interface to the tape drive `ioctl()` calls. For example, the `ioctl()` operation to load a tape is `MTLOAD`. `TarQueue` calls this operation through the `mtio.load()`. The `mtio.h` header file defines the macros and structures for the `ioctl()` tape operations, including bit masks for the drive's generic status word. The simple program `mtio.c` prints the macro values and bit masks as integer constants and functions in Python syntax. The `mtioctl.py` module is generated directly from the output of the `mtio` command.

### 4.2.3. Process Control

To properly interrupt the `select()` call and get it to return immediately when the child `tar` program exits, it is necessary to register a signal handler for the `SIGCHLD` signal. `TarQueue` registers a simple function `sigchld_handler` which does nothing:

```
signal.signal(signal.SIGCHLD, sigchld_handler)
```

This overrides the default behavior of ignoring `SIGCHLD` and causes `select()` to return with the interrupted error `EINTR` when `SIGCHLD` is received. However, there is still a race

condition in which the signal might arrive outside the `select()` call, thus without a timeout the `select()` could block indefinitely. Instead the `select()` is passed a timeout period, and the exit status of a child process is checked with `waitpid()` on every iteration of the event loop, just in case the `SIGCHLD` was missed.

### Note

For details about signals and the race condition, see the Linux `select()` man page.

#### 4.2.4. Other implementation notes

GNU tar exits with an exit code of 2 if one of the files on the command line could not be read, same as it does for just about any other error, so we need to be careful to remove any files from the queue which do not exist. Otherwise, every attempt to tar files would fail and the queue would back up.

### 4.3. Input and Output Catalogs

The `TarQueue.py` module must be setup with the file paths of two catalogs, one acts as an input queue and the second acts as an output queue. As file entries in the input queue are successfully written to tape, the file entries are moved from the input queue to the output queue. For an explanation of file catalogs, see Section 6.

The `spol_tar` script runs a `TarQueue` using SPOL-specific, hardcoded paths:

#### `spol_tar` Catalog Directories

`/data/archive_spol/spol_archiver-tar.catalog` This is the input catalog. `spol_archiver` writes file entries into this queue as it creates them.

`/data/archive_spol/spol_tar_done.catalog` This is the output catalog. Once the `tar` process successfully writes a file to tape, `TarQueue` moves the file entry from the input catalog to this output catalog.

## 4.4. Status Information

First of all, a `TarQueue` is always in one of the following states. The names given here are the exact names returned in the `status` member of the status structure returned by the `getStatus()` method.

### TarQueue States

waiting	The tape device is open but nothing is happening because nothing is in the queue.
tar-in-progress	A <b>tar</b> process is currently running.
load-media	This is the state whenever <code>TarQueue</code> needs to open the tape device. <code>TarQueue</code> enters this state when it first starts, and it remains there until the tape device is opened successfully by the <code>loadMedia()</code> method. It also returns to this state when the current tape needs to be swapped.
unloading	When an <code>unloadMedia()</code> request is made while a <b>tar</b> is in progress, the status will be unloading. <code>TarQueue</code> will not unload the device and close it until the <b>tar</b> process finishes. Once there is no pending <b>tar</b> process, the device will be unloaded and closed and <code>TarQueue</code> will enter the waiting status.

Whenever a **tar** process fails, **TarQueue** assumes the tape is full, closes the tape device, and re-enters the load-media state.

In the load-media state, the `mediaID` field of the status holds the identifier which has been generated for the next tape. A GUI controller can prompt the operator to label the tape with this ID before loading it into the drive.

## 4.5. XML-RPC Methods

`TarQueue` also runs a `SimpleXMLRPCServer` on port 35306 to respond to status requests and to provide a simple handshake with a GUI for swapping tapes. Several remote method calls are available:

### TarQueue XML-RPC Methods

<code>getStatus()</code>	Returns the current status structure.
--------------------------	---------------------------------------

<code>processQueue()</code>	This is the main entry point for checking the input queue and assembling the queued files into a <b>tar</b> command. This method is called periodically from the <code>TarQueue</code> event loop (method <code>run()</code> ), according to the interval set by <code>setInterval()</code> . The interval defaults to one minute.
<code>quit()</code>	Exit the event handler loop. There is no attempt to wait for a current <b>tar</b> process to finish and no attempt to unload the current tape. The <b>spol_tar</b> script just exits when the <code>run()</code> method returns.
<code>loadMedia()</code>	Attempt to open the device path. If successful, the new state will be waiting, and subsequent triggers of the <code>processQueue()</code> method will cause any files in the queue to be written to the open tape.
<code>unloadMedia()</code>	Close the tape device so the tape can be unloaded. The status changes to load-media.

## 4.6. Tape Format

**spol\_tar** limits the number of data files per tar file on the tape to one, the current default in `TarQueue`. Every data file is preceded in the tar archive by its XML file entry from the input catalog.

`TarQueue` writes an extra EOF to the tape after every tar file. If something different is desired, it is easy to change.

## 4.7. Verifying Tapes

There is a simple script to check a tape while the files are still online: **spol/archiver/restore\_and\_check**. It just untars files from the tape device and compares them against the originals in `/data/archive_spol`. Once the MD5 checksums are included in the catalog entries by the disk archiver, the script can check the integrity of tapes without the data files remaining online by comparing checksums. The checksum can come either from the XML file written on the tape, or from the tape archiver's output catalog.

## 4.8. Batch Control with xquery

It is possible to control `TarQueue` from the command line or from a batch script using an XML-RPC client to send method calls to the `TarQueue` XML-RPC server. In particular, the `raddx/spol/archiver` source directory contains just such a client called **xquery**.

When run without any arguments, **xquery** prints out its usage information:

Usage: `xquery serverPort [method] [serverHost]`

At a minimum, **xquery** requires a port number for the XML-RPC server to contact. The *method* defaults to `getStatus` and the *serverHost* defaults to `localhost`. Therefore since the default port for the `TarQueue` status server is 35306, this command queries `TarQueue` and prints the current status as the raw XML return value:

```
xquery 35306
```

Note that `TarQueue` always logs the address of its XML-RPC server at startup, so check for a line like the following to find the particular port and host of a `TarQueue` instance:

```
XML-RPC server at localhost:35306
```

Since `TarQueue` does not automatically load a tape on startup, then to run it from the command line it must at least be told to load the tape. This is the effect of the `loadMedia` XML-RPC method mentioned in Section 4.5. Here is the corresponding **xquery** command:

```
xquery 35306 loadMedia
```

Likewise all of the other XML-RPC methods can be called through **xquery**. The effect of the calls is the same as if they had been called from the GUI controller.

## 5. GUI Status and Control

**rda\_status** has two buttons for opening archiving status: 'Combined Disk Archiver' shows the archiver status on port 32005, such as throughput and disk space remaining, and 'Tape Archiver' connects to **spol\_tar** on port 35306. The tape archiver button turns yellow when a tape needs to be loaded.

Following is the expected operations sequence between the tape controller and the tape archiving process itself:

Upon startup, **spol\_tar** will not write anything, to prevent overwriting a tape left in the drive after a previous crash or stop. It starts in the load-media state, so its **rda\_status** button will be yellow. Click on the button. The status window shows a media ID, the ID which will be assigned to the next tape to be loaded. Probably the format will change to something more conventional, but for now it is just a timestamp. The operator can label the tape before

loading it, then click on 'Load Tape' button. The 'Load Tape' button calls the `loadMedia()` method on `TarQueue`, so `TarQueue` opens the tape device. Once the tape device is open, subsequent scans of the queue will start writing files to tape with **tar**. The queue is scanned every minute by default, but that can be changed in the **spol\_tar** script by calling the `TarQueue.setInterval()` method. Each file entry moved to the output queue is augmented with the media id to which the file was written.

The throughput stats in the tape archiver status window are not implemented yet and will always be zero.

When a **tar** child process exits with an error, `TarQueue` assumes either the tape is full or it otherwise needs to be replaced, so it unloads the tape and returns to the load-media state.

## 6. File Catalogs

The disk and tape archiving processes must carefully coordinate the migration of files through several steps, from being created by the disk archiver, to being written to tape, and finally to being scrubbed from the disk to make room for more files. To help this coordination, each data file is inserted into a *file catalog*. The file catalog simply contains an entry representing each data file in the catalog, and each entry contains information related to the data file and its processing state. A catalog can act like a queue: one process inserts file entries into a catalog while a second process acts on those files then removes the file entries. Note that the data files themselves do not move and are not altered in any way, only the catalog entries move from catalog to catalog.

The file catalog is implemented as a directory; the directory name always ends with `.catalog`. An entry in the catalog is implemented as an XML file in the catalog directory with a `.xml` extension. Thus catalog entries can be inserted and removed from a catalog atomically, using the filesystem `rename()` function. The XML file for a catalog entry contains the meta-information about that file, such as when it was created, its size, and whether the file has been written to tape. Since data files and catalog entries are separate, the data files themselves can be scrubbed from the system, but a catalog can still hold a record of that data file and when it was scrubbed. Likewise, a data file can be inserted into more than one queue at a time, such as then there are multiple downstream processors.

Using XML for the file entries themselves offers a few advantages. First of all, it makes the information in the files very transparent. Without knowing XML it is possible for someone to interpret the information in an XML file, and since the XML is simple text, it is possible to write scripts or programs to extract basic information from the catalog entries. For more complete parsing and information manipulation, programs can use the standard XML-DOM API available in probably every language. For example, the SPOL disk archiver uses a C++ interface to write catalog entries, while the SPOL tape archiver uses the python `xml.dom.minidom` API to read and extend those same catalog entries. XML helps bridge the gap between what humans can interpret and what machines can parse.

Another advantage to XML is the existing processing tools. Since the collection of file entries in a catalog is essentially a single XML document, XML processing tools like XSLT and CSS can be used to render the information in the catalog in various forms, especially HTML. There are also tools for querying XML documents and extracting subsets of information from them.

The file catalogs used by the SPOL disk and tape archiving are actually a specific application of an *XML object catalog*. The `xmlObjectCatalog` class in the `raddx/domx` library implements an object catalog for instances of `xmlObject`. An `xmlObject` represents its state as an XML DOM document, allowing it to be translated to and from files on disk easily and also to be extended with new attributes and interfaces without losing existing information. The file entries in a file catalog are instances of an `xmlObject` subclass: `xmlFileObject`.