S4P Users Guide

A guide for users of the Simple, Scalable, Script-Based, Science Processor (S4P)

September 2008

Document Version 1.0.0

Chris Lynnes, NASA
Stephen Berrick, NASA

Table of Contents

1. Introduction

Data processing costs too much. This fact was brought home during the development of the systems to process data from the Terra platform. Integration of the science algorithms has turned out to be more difficult than expected, leading to higher costs, reduced capability and schedule slips. This has motivated an evolution in the Earth Observing System Data Information System (EOSDIS) toward Science Investigator-led Processing Systems (SIPS). In mid-1998, the Goddard Earth Sciences Data and Information Services Center (GES DISC) began developing a simplified processing system as a risk mitigation and a potential resource for future SIPS.

Many science processing systems have simply grown up around the algorithms they run. Although simple and robust, they often are specific to the algorithm. On the other hand, the EOSDIS Core System (ECS) was designed to be general, resulting in a large, complex mix of commercial and custom software. Recent successful large systems include the SeaWiFS Data Processing System and the TRMM Science and Data Information System. While developed for specific disciplines, they are in fact relatively easy to generalize to other algorithms. One thing that these have in common is the use of commercial databases (often Sybase), and in most cases commercial system management tools (*e.g.* AutoSys).

In contrast, many simpler systems, such as the EROS Data Center AVHRR 1KM system, rely on a simple directory structure to drive processing, with directories representing different stages of production. The system passes input data to a directory, and the output data is placed in a "downstream" directory.

The GES DISC's Simple, Scalable, Script-based, Science Processor (S4P) is based on the latter concept, but with modifications to allow varied science algorithms and improve portability. It uses a factory assembly line paradigm: when work orders arrive at a station, an executable is run, and output work orders are sent to downstream stations.

1.1 Design Goals

Based on the above premises, the following design goals were used in the development of S4P as well as all the projects based on the S4P core (such as S4PM and S4PA):

1.1.1 Simplicity

In all code, a concerted effort must be made to keep it simple. Non-comment lines of code should be kept as small as possible without becoming cryptic. (Comments, on the other hand, should be used liberally.)

1.1.2 The 80/20 Rule

There is a rule of thumb that one often achieves 80% of the functionality with the first 20% of the effort. This tells us to move on to the next item when we have achieved that 80% functionality, coming back for more only if time permits.

1.1.3 "Use the OS, Luke!"

Years and years of development have gone into the operating system which is optimized for the machine it is on. Thus, if you can use the operating system to achieve something for you (*e.g.* files, directories, UNIX commands), do not bother to reinvent it. The only exception is that if you know with absolute certainty that (a) the OS solution will not meet performance requirements and (b) your solution will.

1.1.4 Design-For-Trouble

Things will go wrong in any system. Therefore, the design should have features to enable troubleshooting built in from the start, not simply added on when things don't work. This does not necessarily mean automated failure recovery, simply ensuring that human operators can easily and quickly determine what has gone wrong.

1.1.5 Transparency

Following from the Design-for-Trouble principle, the operation of the system should be as transparent as possible. That is to say, operational staff and sustaining engineers should be able to see easily everything that is going on, the contents of data moving hither and yon, etc. (This is the principle behind the use of work order files, rather than socket messages, which can disappear into the ether.)

1.1.6 Keep Things Together

Troubleshooting is easier if everything (*e.g.,* input, output, templates, configuration files, logs, etc.) can be found in one place.

1.2 S4P Features

Below are listed a number of features of the S4P system.

1.2.1 Station (formerly, Stationmaster)

Station is the linchpin of the S4P. It is a refactored version of and replacement for Stationmaster (the old Stationmaster script, stationmaster.pl, should be considered deprecated and will likely be removed in a future release). The s4p_station.pl script that implements Station has a number of configurable features to provide the necessary flexibility to construct any arbitrary processing system.

- Adjustable polling interval
- Run different executables based on job type
- Child process limits
  - Total number of running and failed children
  - Number of running children
  - Number of failed children (will stop forking jobs when too high)
- Send output work orders to multiple downstream stations
- Send output work orders to different downstream stations based on output job type
- Run a user-defined failure analysis program on job failure, based on input job type
- Maintain logs of the full history of a given work order (including logs for upstream stations)
- Job sorting and prioritization:
  - By work order name
  - By arrival time (FIFO)
  - By input job type
  - By custom function

1.2.2 Monitoring Stations

Two tools, the S4P Monitor and the Job Monitor work together to monitor a system of stations and control individual stations and jobs.

- Monitor a group of stations for running, failed and pending jobs
- Start and stop a station
- Job Control
  - Terminate a job
  - Suspend a job
  - Resume a job
  - Restart a failed job

In addition, station-specific scripts of various types can be developed and hooked into the S4P Monitor and Job Monitor interfaces. This is done simply by associating a Button name with a command-line executable in a configuration file.

- Failure Handlers work with failed job directories, providing a way to recover from or otherwise handle a failure case. This is particularly useful for specific cleanup tasks.

- Manual Overrides work with currently running directories. These are used, for example, to release jobs that are waiting for something. (Note that this requires the scripts to do some signaling to each other in the job directory.)
- Station-specific interfaces can be attached to any station, allowing access, for example to additional GUIs.

2. Architectural Overview

2.1 Assembly-Line Paradigm

The process of using science algorithms to create products is modeled as a factory assembly line, with a number of **stations** along the way. Two key improvements on a real factory are: (1) multiple instances of a job can be executed simultaneously at a given station, and (2) the output can be sent to more than one downstream station simultaneously.

Simply stated, a working station is a Unix or Windows directory with 2 key components:

1. A configuration file (normally named `station.cfg`). The two essential items in the configuration file are a map of tasks to execute on arrival of input **work orders,** and a map of downstream stations to which output work orders are sent.
2. A daemon monitoring the station for input work orders. A program called **Station** (actually, `s4p_station.pl`) is provided for this purpose; all that need be done is to start up a Station for each station.

Once the set of stations has been established and configured, the system is started by running Station in each station of the system. When the first input work order is deposited in the upstream station, the Station program in that station does the following:

1. Detects the work order
2. Looks up the appropriate command to run for that work order type
3. Makes a temporary subdirectory for executing the job
4. Changes directory into the that subdirectory
5. Forks off a child Station process to run and monitor the command
6. When done, it looks up the output work orders in the downstream map
7. Sends work orders to the downstream stations

2.2 Station Components

A station in S4P is a directory within which a Station daemon is running. The types of work orders that Station daemon looks for and what it does are dictated by configuration parameters in the station configuration file. By default, this file is named station.cfg and this will be the assumption throughout this document. In addition, other behaviors and characteristics of the station are set in the station configuration file such as the name of the station that gets displayed in the graphical user interface, the polling frequency, and many others. Section 4 on the Station program describes the options available in the station configuration file.

Beyond the Station program itself, there are several important components of S4P stations and these are discussed below.

2.2.1 Work Orders

Work orders provide the input for a job in the station. Each work order is processed in turn (there are ways of configuring the sort order). Work orders are placed in the station by upstream stations or via some other mechanism outside of S4P. There are no restrictions on the content of work order; it can be what ever the code running in the station expects. By convention, however, work orders should be ASCII text files since transparency is a major tenet of the S4P philosophy. If you see a work order in a station directory, it either means that a Station daemon isn't running or that the work order simply hasn't been processed yet.

By default work order file names have four components separated by a period character: a prefix, a job type, a job identifier, and a suffix. Thus, work order file names have this form:

```
<prefix>.<jobtype>.<jobid>.<suffix>
```

Work Order Name Component	Description
Prefix	By default, work order file names have DO as a prefix. See Section 4.4 for options on overriding this default
Job Type	The job type dictates what to run when a work order having that type shows up in a station. It can be any ASCII string of characters not including the period character (.). By convention, job types are all in uppercase. Since job types dictate what will be run, it is wise to keep job types short and descriptive.
Job ID	A job ID is used to guarantee uniqueness in work order file names. Typically, it is the machine time or perhaps the process ID. The only requirement is that it be unique for each work order of a given type.
Suffix	By default, work order file names have wo as the suffix (short for "work order"). As with the prefix, Section 4.4 on the Station program describes some ways to modify this default.

Table 2-1. S4P work order file name components.

2.2.1.1 Example Work Order Names

For example:

```
DO.IMPORT.398983.wo
```

has a job type of IMPORT and a job ID of 398983 (we presume that it is unique among all work orders of type IMPORT).

```
DO.RUN_ALG1.2006123201030503.wo
```

has a job type of RUN_ALG1 and a job ID of 2006123201030503 (which looks like a timestamp).

2.2.1.2 What Goes Into a Work Order

S4P doesn't actually care about what content there is in a work order. The only thing S4P cares about is the name of the work order since that is how it decides what to run on it. Empty work orders are perfectly valid too (though perhaps not useful).

2.2.2 Log Files

2.2.3 Station Log

There are several log files in S4P. By default, the Station program running in each station maintains a log file named, by default, station.log (to override this, see Section 4.7.2). This file lists out work orders processed and notes any failures. All messages are time tagged.

2.2.4 Chain Log

In addition, any output to standard out (stdout) and standard error (stderr) generated by the commands running in the station (*i.e.* the processing code) is saved into a log file whose name looks like the work order name with .log as the file name extension. For example, if the work order name was:

```
DO.RUN_ALG1.4983.wo
```

the log file generated would be named:

```
RUN_ALG1.4983.log
```

Two functions are provided with the S4P distribution to facilitate logging of messages to the chain log file. They are S4P::logger() and S4P::perish. The S4P::logger() function takes two arguments. The first is a tag indicating the level of severity of the message (valid choices are "INFO", "WARNING", "ERROR", "FATAL", and "DEBUG") and the second is the message string itself. The message written out to the log file will include a timestamp and the name of the Perl script generating the message. Note that messages tagged with "DEBUG" only get written if the environment variable OUTPUT_DEBUG is set to non-zero; otherwise they are ignored.

The S4P::perish() function also writes out a message to the chain log file, but then exits (dies) afterward. It also takes two arguments. The first is the exit status with which to exit and the second is the message string. As with S4P::logger(), messages are written along with timestamps and the scripts generating them.

If the processing in a station generates any output work orders destined for downstream stations, the log file produced by the processing code gets moved downstream along with

the work order itself. New log messages will get appended to the log file. Thus, the log file grows as the work order with which it is associated is passed from station to station, hence, the name "chain log".

2.2.5 Processing Code

Of course, the most important component of any S4P system is the code that actually carries out the processing in each of the stations making up a string. As suggested by earlier examples, there are very few restrictions on what processing can be done in S4P. Below are the assumptions that S4P processing code must meet:

1. S4P interprets an exit code of non-zero to be a failure and an exit code of zero to be success. Thus, anything that S4P runs in a station needs to adhere to this convention.
2. The file name of the work order is automatically appended to the end of the command being run. Thus, if the processing code is a script or compiled binary, it should be prepared to receive the input work order as the last argument. There is no requirement that the code actually open or read the work order; it can ignore it altogether.
3. For processing code that does read the input work order name, that file name follows a convention described in Section 4. By default, input work orders have a 'DO.' prefix.
4. Processing code may generate one or more output work orders (or none at all). Output work orders must have (by default) the '.wo' suffix and must **not** have the 'DO.' prefix. See Section 4 for ways of changing this default.
5. There are no restrictions on the actual content of work orders. For the sake of transparency (a tenet of S4P philosophy), ASCII work orders are much preferred over binary.
6. Processing code needs to be aware that it is running within a unique subdirectory below the station directory. This can be important if processing code needs to access other files (aside from input work order) or directories. The unique subdirectory is removed once the job completes, therefore, processing code cannot assume that the job directory is permanent. If files (such as databases) need to be maintained across runs, they should be placed in the station directory itself or nearby.

2.3 Processing A Work Order

When the Station program detects a work order in its station directory following the naming convention discussed in Section 2.2.1, it creates a subdirectory with the name of the work order prefixed with 'RUNNING'. For example, the work order:

```
DO.IMPORT.200612140122.wo
```

would get a subdirectory named:

```
RUNNING.IMPORT.200612140122
```

That is, the job subdirectory name is taken from the work order name with the 'DO' prefix replaced by 'RUNNING' and the suffix 'wo' removed.

Next, the Station program moves the work order into the new subdirectory created for it, but at the same time dropping the 'wo' suffix. Thus, the work order within the job subdirectory would actually be:

```
DO.IMPORT.200612140122
```

This is important to keep in mind since the process running in the station on the work order needs to not expect the 'wo' suffix in the file name. It won't be there.

Once the work order is in the job subdirectory, the Station program forks off a child Station process to execute and monitor that job.

If the job fails, the subdirectory name prefix 'RUNNING' is changed to 'FAILED' as in:

```
FAILED.IMPORT.200612140122
```

This allows one to easily spot failures by simply doing a directory listing in a station directory.

If the processing succeeds, there may be one or more output work orders generated as a result. Output work orders must follow this convention:

```
<jobtype>.<jobid>.wo
```

as in:

```
RUN_ALGORITHM_1.200612140122.wo
```

This is somewhat opposite of input work order. For input work orders, the original 'wo' suffix is dropped, but the 'DO' prefix is kept. For output work orders, the original 'DO' prefix is dropped and instead the 'wo' suffix is kept. Processing code in stations that generate output work orders need to keep this in mind when setting their file names.

The Station program will see to it that output work orders are passed on to downstream stations (if any). We'll see how to specify this later. In summary:

Work Order Flavor	Work Order Example
Work order in a station directory queued up to be run.	DO.IMPORT.200612140122.wo
Input work order in a job subdirectory being processed.	DO.IMPORT.200612140122
Output work order in job subdirectory after processing has completed.	RUN_ALGORITHM_1.200612140122.wo

Table 2-2. S4P work order flavors and examples.

2.4 Station Monitoring

Since a S4P processing string is nothing more than directories, subdirectories, and files, it is very easy to monitor S4P processing by simply listing out the contents of station directories. As an illustration, running the command:

```
ls -F import
```

on a fictitious Import station directory (in UNIX), might produce something like this:

```
import.pl*
station.cfg
station.lock
station.log
station.pid
station_counter.log
DO.IMPORT.2006121223221.wo
DO.IMPORT.2006121223342.wo
IMPORT.2006121223221.log
IMPORT.2006121223342.log
RUNNING.IMPORT.2006121222020/
RUNNING.IMPORT.2006121221821/
FAILED.IMPORT.2006121172927/
```

Immediately, one can see that there are two jobs running, one job has failed, and there are two work orders representing two jobs queued up to be run. You probably have guessed already that the import.pl file is a Perl script that is run to carry out the processing in our imaginary Import station. The station.pid and station.lock files simply contain the process ID of the Station process running in this station (some operating systems have a preference for the PID file; others for the lock file). We have not yet talked about the station_counter.log file.

If you were to change directories into the FAILED.IMPORT.2006121172927 subdirectory, you would see the debris left there at the time the job failed. By examining the log files therein, hopefully the problem causing the failure could be resolved.

Because of the simplicity of this structure, it is fairly easy to develop a graphical user interface (GUI) to display the state of multiple S4P stations comprising an S4P string at any instant in time. In fact, S4P include a Perl/Tk implementation of just such a GUI. The programs tkstat.pl and tkjob.pl will be discussed later in Section 5 on S4P interfaces.

3. S4P Installation

This section describes how to download and install S4P. This document is current as of S4P 5.15.0.

S4P made available on SourceForge as part of the S4PM distribution at http://sourceforge.net/projects/s4pm/.

3.1 Installation Requirements

S4P has been successfully installed on SGI machines running IRIX, Sun machines running Solaris, and Linux machines running Red Hat. It should work on any UNIX machine. S4P has also been tested successfully on a Window XP machine (using ActiveState Perl), but the testing here has been very light and recent versions of S4P have *not* been regression tested. We would appreciate hearing from those of you who have run S4P under Windows.

S4P requires Perl 5.6.0 or later (it *might* work with earlier versions). It also requires the Perl Tk module.

3.2 Basic Installation

There is only one package to download:

S4P-5.15.0.tar.gz

Download the package into some directory on the machine where you will install S4P. Version 5.15.0 is assumed in the examples below. Adjust the version portion of the file names according to the version you are using.

The directory you download this package into is only used for installing S4P and can be removed later.

Unzip and untar the package file. On Linux, you can untar and unzip with one command:

```
tar xvzf S4P-5.15.0.tar.gz
```

On other UNIX machines, you may have to unzip and untar separately:

```
gunzip S4P-5.15.0.tar.gz && tar xvf S4P-5.15.0.tar
```

Unpacking this tar file will result in one subdirectory: S4P-5.15.0

Change directories into the S4P-5.15.0 directory:

```
cd S4P-5.15.0
```

For installation of the binaries into the standard system directories on your machine, run the following:

```
perl Makefile.PL
make
make test (recommended, but optional)
make install
make clean (optional)
```

3.3 Customized Installation

You can choose to install S4P into a non standard location. This means that you will need to specify where the binaries and Perl library modules go directly.

First, you'll first need to set the environment variable PERLLIB (on Linux, use PERL5LIB instead) to the alternate location of the libraries. Both the S4P and S4PM libraries will need to be included (see example below) and you will need to set this environment variable ***before*** you build S4P. The PERLLIB (or PERL5LIB) environment variables will also have to be set correctly in order to run S4P. Finally, run the following commands instead of the ones above:

```
perl Makefile.PL PREFIX=<alternate_directory>
make
make test
make install
make clean (optional)
```

where <alternate_directory> determines both where the binaries and libraries are to be installed. The binaries (scripts and configuration files) will be installed in <alternate_directory>/bin; the S4P libraries will be installed into <alternate_directory>/lib/perl5/site_perl/perl5.x.x/. See example below.

Also, if you install the binaries into a non standard directory, the user account under which S4P will be run will have to include this new location in the PATH environment variable.

3.3.1 Customized Installation Example

For example, you wish to install the S4P binaries and libraries under /home/jdoe rather than in the standard system directories. Follow these steps:

1. Log in as the S4P user that will run S4P.

2. Set the PERLLIB (it may be PERL5LIB if Linux) to where the libraries are to be installed. For example (in Bourne, Korn, Bash shell or their variants):

```
export PERLLIB=/home/jdoe/lib/perl5/site_perl/5.8.3
```

3. Run the install:

```
perl Makefile.PL PREFIX=/home/joe
make
make test
make install
make clean (optional)
```

4. In the S4P user's shell start up scripts (e.g. .bashrc), set the PERLLIB or PERL5LIB environment variable as above and also set the PA environment variable to include /home/joe/bin.

The S4P components will be installed into the directories as indicated in Table 3-1 below.

Component	Installation Directory
Executable Scripts	/home/jdoe/bin
Configuration Files	/home/jdoe/bin
S4P Perl Library Modules	/home/jdoe/lib/perl5/site_perl/perl5.8.5

Table 3-1. S4P components and where they would go if alternate location is set to /home/jdoe. Here, we assume a Linux installation using Perl 5.8.5.

3.4 What's Included in S4P?

S4P is comprised of Perl scripts (*.pl files) and Perl modules (*.pm files). Below is a brief summary of most of what's included. For more details on these and other script and modules, read the man pages or look at the code itself.

S4P File	Description
remove_job.pl	Removes a failed job.
restart_job.pl	Restarts a failed job.
s4p_repeat_work_order.pl	Wrapper script that allows a station to automatically run a job on a periodic basis by continually recycling work orders.
s4pshutdown.pl	Shuts down all stations in a S4P string.
send_downstream.pl	Sends a work order to the next station downstream.
s4p_station.pl (formerly, stationmaster.pl)	Runs the Station daemon on a station to poll for incoming work orders, start up processing on those work orders, and move any output work orders downstream.
stop_station.pl	Stops a station.
tkjob.pl	Perl/TK GUI that allows a user to examine the contents of a job directory.
tkstat.pl	Perl/TK GUI that allows a user to visually monitor activity in multiple S4P stations and perform troubleshooting.

Table 3-2. Some of the S4P scripts included.

S4P Module	Description
S4P	Main S4P Perl module with the core S4P functionality.
S4P::EDOS	Contains functions for parsing Level 0 construction records that adhere to the CCSDS format.
S4P::FileGroup	Along with the S4P::FileSpec module, handles manipulations of Product Delivery Records (PDRs), a type of file implemented in Object Data Language (ODL), a precursor to XML. Used mainly for sites running the EOSDIS Core System (ECS) or interoperating with ECS via standard Science Investigator-Led Processing Systems (SIPS) interfaces.
S4P::FileSpec	See description for S4P::FileGroup.
S4P::Lexer	Provides a generic lexer class.
S4P::MetFile	Contains functions for manipulating ODL metadata files that are used in the EOSDIS Core System (ECS).
S4P::OdlBlock, S4P::OdlGroup, S4P::OdlObject, S4P::OdlTree	These modules are used for manipulating ODL metadata via a data tree model.
S4P::PAN	Contains functions for manipulating ECS-style Product Acceptance Notifications (PANs).
S4P::PCF, S4P::PCFEntry	Contains functions for manipulating ECS-style Process Control Files (PCFs).
S4P::PDR	Contains functions for manipulating Product Delivery Records (PDRs).
S4P::ResPool	Contains functions for creating and using a disk pool management system, useful for S4P strings that need to manage disk space. Allocation and de-allocation of disk pools is provided.
S4P::S4PTk	Contains functions used by S4P GUIs.
S4P::Station	Contains functions for supporting Station and Job objects.
S4P::StaMon	Module used by tkstat.pl for creating a station monitor object.
S4P::Subscription	Contains functions for managing subscriptions for online data.
S4P::TimeTools	Contains many time manipulation functions.
S4P::TkJob	Module used by tkjob.pl for creating a job monitor object.

Table 3-3. Some of the S4P modules included.

4. The Station Program

The Station program is the driver of S4P. It monitors station directories for incoming work orders, runs configured processing on those work orders, and moves any output work orders to downstream stations that are monitored by their own Station damons. The Perl code that implements Station is s4p_station.pl. This new program replaces what was formerly called Stationmaster (stationmaster.pl).

Note that the –C option for s4p_station.pl will emulate stationmaster.pl behavior. This same option is available for tkstat.pl as well. The option will likely be dropped at some point once the old stationmaster.pl script itself is dropped.

An instance of Station monitors only a single station where a station is simply a directory containing, at a minimum, a station configuration file that dictates how the Station should behave. By default, this configuration file is named station.cfg. If there are a series of Station programs running on a series of stations and if these stations pass work orders to one another, the group of stations is often referred to as an S4P "string".

4.1 Running Station

Station is run on the command line via this command:

```
s4p_station.pl &
```

In UNIX, the ampersand forces the process (s4p_station.pl, in this case) to run in the background. By default, the Station program will monitor the current directory. That is, it assumes that the current directory is the station directory and contains a station configuration file named (by default) station.cfg.

To specify the station directory explicitly, use the –d option along with the full or relative path to the station directory as in:

```
s4p_station.pl –d stations/import &
```

There are several more rather esoteric options available to Station and the interested reader can discover them in the s4p_station.pl man page .

If there are more than several stations, typically one starts up the S4P Monitor and through that interface starts up the individual stations. The S4P Monitor will be discussed in Section 5.

4.2 Station Configuration Basics

In this section, the most basic station configuration file parameters are discussed:

Parameter	Description	Section
%cfg_commands	What to run for each work order type	4.2.1
%cfg_downstream	Where to send any downstream work orders	4.2.2

Table 4-1. Basic station configuration file parameters.

4.2.1 Commands To Run (%cfg_commands)

The %cfg_commands is the most important parameter in the station configuration file since it defines what to run for each work order type that is detected by Station. As mentioned earlier, each station much be configured for particular work order types.

Work orders that have types other than those defined in the station configuration file will be placed (by default) in a subdirectory under the station directory named FAILED.WORK_ORDERS and will be visible through the S4P Monitor (Section 4) as a red job box. If the entire pattern of the work order file name is not recognized, however, it will be ignored.

The hash keys of %cfg_commands are work order types and the hash values are commands that Station should run when a work order of that type shows up in the station directory.

The work order itself is always automatically passed to the command as the last argument.

Below is an example of how to use %cfg_commands:

```
%cfg_commands = (
        'IMPORT' => '../import.pl',
);
```

In the above example, the Station program is configured to execute '…/import.pl' when ever a work order of type IMPORT is detected. Remember though, the actual work order name is passed implicitly as the last argument to what ever is set in this hash. It is up to the script, import.pl in this case, to make use of the work order contents in whatever way is appropriate (it can even ignore it). Thus, the actual command run is this:

```
../import.pl workorder_filename
```

where `workorder_filename` is the actual name of the work order. Such work orders would look like this:

```
DO.IMPORT.3982989
```

Note that the .wo file name extension is missing. Why? The reason is that although work orders by default all start with DO. and end with .wo, once the work order is moved into a station subdirectory for running, the .wo file name extension is dropped by Station. Thus, it is important that station scripts don't rely on the .wo file name extension because it won't be there.

The command to run for a particular work order type doesn't have to be a script or a binary executable. It can be a UNIX command such as:

```
%cfg_commands = (
        'CLEAN' => "find /data -type f -mtime +8 -print -exec /bin/rm
{} \\; && echo'"
);
```

The above rather complicated command will run the UNIX find command to clean out data files that order than eight days.

Also note that the executable, import.pl, is specified with a relative path (the `../`). A S4P tradition is to place station scripts (or symbolic links to them) in the station directories. Since a job running in S4P runs in a subdirectory *under* the station directory, the script is always one directory up. If, however, station scripts are locatable in the user's path, this is not necessary.

More than one work order type can be configured in a station. For example:

```
%cfg_commands = (
        'IMPORT' => '../import.pl',
        'RESYNC' => '../resync.pl -d ../database.db',
);
```

The station configured with the above setting will recognize two types of work orders and run a unique task depending upon which type is detected. In each case, the work order file is always passed as the last argument to the command. Thus, for a work order of type RESYNC, the actual command run is:

```
../resync.pl -d ../database.db workorder_filename
```

Note that in this example, if a work order shows up in this station with, say, the name DO.RUN.93839.wo, the basic pattern will be recognized by Station (it starts with a DO and ends with a wo), but it will end up in the FAILED.WORK_ORDERS subdirectory since its type is not defined in the %cfg_commands hash. If, however, a work order with the name RUN.98393.wo shows up, it will be ignored since it doesn't even qualify as a bona fide work order (it's missing the wo suffix).

4.2.2 Downstream Work Orders (%cfg_downstream)

After a station processes an input work order, it may generate one or more output work orders to be sent to other stations. A station is not required, however, to generate any output work order.

Station uses another hash in the station configuration file to decide where output work orders go. Below is an example:

```
%cfg_downstream = (
        'RUN_ALG2' => ['../../run_algorithm_2'],
);
```

The above hash tells Station that output work orders of type RUN_ALG2 found in the job directory after the processing has been completed successfully should be routed to the station whose directory name is ../../run_algorithm_2. Note that since this is being viewed from within a job subdirectory under the station directory, the relative path needs to include the ../.. (this assumes that the run_algorithm_2 station directory is at the same level as the current station directory. Later on, we'll discuss a way of setting a default station root so that this won't be necessary.

It is up to the script running in a station to generate work orders of the correct type and with the correct file names. Output work order file names must follow this convention:

```
<jobtype>.<jobid>.wo
```

Note that there is no DO. in front. Thus, given the definition of %cfg_downstream above, a work order that looks like this:

```
RUN_ALG2.983983.wo
```

Station will detect this output work order, add the DO. suffix and move it into the run_algorithm_2 station directory.

If a station generates more than one work order, it is easy to accomplish:

```
%cfg_downstream = (
        'RUN_ALG1' => ['../../run_algorithm_1'],
        'RUN_ALG2' => ['../../run_algorithm_2'],
);
```

It is also easy to send the same output work order to two or more downstream stations at the same time:

```
%cfg_downstream = (
        'RUN_ALG1' => ['../../run_algorithm_1', '../../track' ],
);
```

In the above example, the output work order RUN_ALG1 is sent to both the run_algorithm_1 and track stations.

Output work orders can be sent to a plain old directory that isn't a S4P station. The Station program won't care:

```
%cfg_downstream = (
        'RUN_ALG1' => ['../../run_algorithm_1', '../../ARCHIVE'],
);
```

Above, the RUN_ALG1 is sent to the run_algorithm_1 station as well as a directory named ARCHIVE that, presumably, saves the work orders for future use.

4.3 Putting It All Together

Below is an example of a complete station configuration file using what we've learned so far for a station named import. This station receives work orders of type IMPORT, processes them with the script import.pl which outputs a work order of type RUN_ALG2, which gets directed downstream to a station named run_algorithm_2. It is named station.cfg.

```
%cfg_commands = (
        'IMPORT' => '../import.pl',
);
%cfg_downstream = (
        'RUN_ALG2' => ['../../run_algorithm_2'],
);
```

4.4 Common Optional Station Parameters

The only truly required parameter in the station configuration file is the %cfg_commands hash. If a station produces no output work orders, the %cfg_downstream hash is not needed.

In this section, we will discuss and provide examples for some common optional parameters that are available for altering the behavior of Station.

Parameter	Description	Section
$cfg_root	Sets station top-level directory.	4.4.1
$cfg_polling_interval	Sets how often Station looks for new work orders	4.4.2
$cfg_stop_interval	Sets how often Station looks specifically for STOP work orders	4.4.3
$cfg_end_job_interval	Sets how often Station looks specifically for END_JOB_NOW, SUSPEND_JOB_NOW, and RESUME_JOB_NOW work orders	4.4.4
$cfg_deadline	Sets maximum time to run a job	4.4.5
$cfg_disable	Disables or enables station	4.4.6
$cfg_input_work_order_suffix	Sets the expected input work order suffix	4.4.7
$cfg_output_work_order_suffix	Sets the expected output work order suffix	4.4.8
$cfg_work_order_pattern	Sets the pattern of the entire work order name	4.4.9
$cfg_max_children	Sets the maximum number of jobs that can be run concurrently	4.4.10
$cfg_user	Sets the userid of the user that is allowed to run the station	4.4.11
$cfg_group	Sets the group of users that are allowed to run the station	4.4.12
$cfg_host	Sets the machine on which the station is allowed to run	4.4.12
$cfg_sort_jobs	Sets the method by which work orders are ordered	4.4.15
%cfg_reservations	Predefines number of work orders per work order type that can run concurrently	4.4.16

Table 4-2. Common optional station configuration file parameters.

4.4.1 Station Root ($cfg_root)

The $cfg_root parameter is an optional parameter that sets the root directory of the stations. This allows one to set the %cfg_downstream hash like this:

```
$cfg_root = '../..';
%cfg_downstream = (
        'RUN_ALG2' => ['run_algorithm_2'],
);
```

rather than like this:

```
%cfg_downstream = (
        'RUN_ALG2' => ['../../run_algorithm_2'],
);
```

4.4.2 Polling Interval ($cfg_polling_interval)

The $cfg_polling_interval sets the Station polling interval in seconds. The default is ten. That is, a Station will look for work orders every ten seconds. One reason to increase the polling interval is if work orders are expected to show up in a station only infrequently or with very long intervals of time in between. Although unnecessary polling by Station has low overhead, on heavily loaded systems, this may be significant.

Very rarely will it make sense to lower the polling interval. If you do, be aware that lowering the polling interval to about three seconds or less may result in "work order poaching". This happens when the time it takes Station to make a job subdirectory, move the work order into that subdirectory, and fork off a Station process to monitor that job takes longer (on occasion) than the polling interval for that station. What happens is that while the Job process is performing these steps, another polling takes place and the very same work order is detected a second time (we've actually seen the same work order get detected three times on a system with a very busy file system!). The result is that the second polling will not find the work order between the time it notices it and the time it's ready to move the work order into the job subdirectory. It's been poached! You end up with a job subdirectory with no work order in it and the Station process forked off for that job fails. So, be cautious.

For example:

```
$cfg_polling_interval = 120;
```

4.4.3 Stop Interval ($cfg_stop_interval)

The $cfg_stop_interval is the number of seconds between polls of the station directory to look for STOP work orders. The default is the $cfg_polling_interval.

Stopping a station in S4P is accomplished via a STOP work order being dropped into the station directory. By default, a STOP work order will only be detected at the polling frequency set for all work orders, the $cfg_polling_interval. When $cfg_polling_interval is very large, stopping a station may take too long. Therefore, a separate more frequent poll can be set up to look exclusively for STOP work orders. The polling interval is $cfg_stop_interval.

For stations whose $cfg_polling_interval is the default (five seconds), it may not be necessary to have a more frequent polling cycle for STOP work orders.

For example:

```
$cfg_polling_interval = 120;
$cfg_stop_interval = 5;
```

4.4.4 End Interval ($cfg_end_job_interval)

The $cfg_end_job_interval sets the number of seconds between polls of the Station child process to look for END_JOB_NOW, SUSPEND_JOB_NOW, and RESUME_JOB_NOW work orders. It works much the same way as $cfg_stop_interval, but applies to the suspend and resume functions. This parameter is only relevant when $cfg_group (Section 4.4.12) is specified; it is ignored otherwise. When $cfg_group is specified, the default is $cfg_stop_interval (Section 4.4.3).

For example:

```
$cfg_end_job_interval = 3;
```

4.4.5 Process Deadline ($cfg_deadline)

The $cfg_deadline parameter is the number of seconds beyond which Station will end the process that it is running. The default is no deadline. This can be useful in stations where the processing should be quick and a significant slow down may indicate a problem. It is recommended that the deadline be set well beyond the expected time to avoid false positives in terms of real problems.

For example:

```
$cfg_deadline = 600;
```

4.4.6 Station Disable ($cfg_disable)

The $cfg_disable parameter simply enables or disables a station. If disabled, Station will not run in that station and the S4P Monitor can be configured to not display the station in the interface. Set $cfg_disable to zero to not disable the station (*i.e.* enable it) and set to non-zero to enable. The default is enabled.

This parameter may be useful in more complex S4P strings to control what stations participate in a string given some configuration.

For example:

```
$cfg_disable = 0;
```

4.4.7 Input Work Order Suffix ($cfg_input_work_order_suffix)

By default, input work orders are assumed to have the .wo file name extension or suffix. To override this default with some other suffix, use the $cfg_input_work_order_suffix parameter. For example:

```
$cfg_input_work_order_suffix = 'pcf';
```

In this example, a work order file named:

```
DO.RUN.98938.pcf
```

would get recognized by Station and run whereas a work order file named:

```
DO.RUN.98938.wo
```

would not.

The $cfg_input_work_order_suffix parameter applies to all work order types in this station.

4.4.8 Output Work Order Suffix ($cfg_output_work_order_suffix)

The $cfg_output_work_order_suffix changes the default suffix on output work orders from .wo to the value set here. For example:

```
$cfg_output_work_order_suffix = 'log';
```

Note that if used with work orders going to another station, the upstream station must be configured to recognize such work orders via the $cfg_input_work_order_suffix.

4.4.9 Work Order Pattern ($cfg_work_order_pattern)

If merely changing the input work order suffix isn't enough (see 4.4.5), the $cfg_work_order_pattern allows one to change the entire default work order name convention for input work orders. It doesn't affect output work order names. This parameter can be particularly useful when the work orders are actually files produced outside of S4P. Set the parameter to a glob pattern that will match all input work orders needed. For example:

```
$cfg_work_order_pattern = 'NEWDATA*.{PAN,PDRD}';
```

In this example, Station will look for work orders that start with NEWDATA followed by the file name extension .PAN or .PDRD. A DO. prefix is not assumed and not expected by Station in this case. Further, a .wo suffix is not assumed and not expected. In fact, if either the DO. prefix or .wo suffix were included in the work order name, Station would not see it.

4.4.10 Maximum Number of Jobs ($cfg_max_children)

By default, a Station will fork off a maximum of five jobs at a time in a station. The $cfg_max_children can be used to increase or decrease the maximum number of jobs (children) running in a station.

Setting $cfg_max_children to one effectively makes the station single-threaded. This could be useful when processing needs to have exclusive locks on resources such as files or databases.

If $cfg_max_children is to zero, rather than forking off a new Station child to run the job, the parent Station runs the job itself.

For example:

```
$cfg_max_children = 10;
```

4.4.11 S4P User ($cfg_user)

The $cfg_user parameter sets the userid of the user authorized to run this station. If the user attempting to bring up the Station program in this station isn't the user set here, Station will fail. This parameter can be useful in preventing inadvertent start-ups of stations. But this should not be viewed as a security measure of any kind. For example:

```
$cfg_user = 'jdoe';
```

4.4.12 S4P Group ($cfg_group)

As an alternative to having an S4P station only run for a particular user (with $cfg_user), a station can be configured to only run for a user belonging to a particular group. This is done by setting $cfg_group rather than $cfg_user. When $cfg_group is set, Station will check to see whether the current user is a member of the group specified. This same checking is also done by TkJob.

For example:

```
$cfg_group = "s4p";
```

4.4.13 S4P umask ($cfg_umask)

The $cfg_umask parameter is often used in conjunction with $cfg_group (4.4.12) to set the umask of files written by Station. The default is 002 which doesn't allow for group readable/writeable files. Note that the specification must be octal.

Also, note that including –u umask when executing s4p_station.pl has the same effect.

For example:

```
$cfg_umask = 022;
```

4.4.14 S4P Host ($cfg_host)

The $cfg_host parameter prevents a Station from running if the host on which it is being started up does not match the value set here. In S4P strings where some stations may be mounts on other machines, this parameter can be useful in preventing a station from being brought up on the wrong machine. For example:

```
$cfg_host = 'frodo';
```

4.4.15 Work Order Sort ($cfg_sort_jobs)

Station processes work orders by default in the same order defined by the shell for listing work order file names. In some situations, it may be beneficial to set the sort order to something else. S4P provides several ways of altering the default sort order

4.4.15.1 FIFO

To set the sort order to first-in-first-out, simply include this in your station configuration file:

```
$cfg_sort_jobs = 'FIFO';
```

This can be useful in situation where it makes sense to process work orders in the order that they show up. The FIFO sort order is particularly useful if groups of work orders need to rendezvous in a particular station at roughly the same time.

4.4.15.2 Fixed Order By Job Type

To explicitly set the sort order by job type, simply list the job types in the order of preference as in:

```
$cfg_sort_jobs = [
        'EXPORT',
        'EXPORT_EPD',
        'EXPORT_BROWSE',
        'EXPORT_PH',
        'EXPORT_FAILPGE'
];
```

In the example above, work orders of type EXPORT are given preference over work orders of type EXPORT_EPD which have a higher preference than work order of type EXPORT_BROWSE, etc. By default, the sort order would have been: EXPORT, EXPORT_BROWSE, EXPORT_EPD, EXPORT_FAILPGE, and EXPORT_PH.

Such sorting may be handy to avoid "starving" work orders that process very quickly; you would move them to the front of the list. Alternatively, you could change the work order type name so that it naturally sorts the way you want.

4.4.15.3 Custom Sort Order Function

A third alternative is a custom written sort function. Simply place the Perl module containing the function in the station directory. Refer to the man page on the Perl sort function for a description of how the function must work. The same rules that apply for Perl sort functions apply for these except that ***the sort function must be prototyped***. This requirement is new for s4p_station.pl (stationmaster.pl did not require this). For example:

```
package MySort;
1;
sub by_strlen ($$) {
     my ($a, $b) = @_;
     return (length($a) <=> length($b));
};
```

Then, set the parameter to the function as in:

For example:

```
$cfg_sort_jobs = "MySort::by_strlen";
```

4.4.16 Work Order Type Reservations (%cfg_reservations)

The $cfg_max_children sets the maximum number of jobs that can run in a station. By default, it is set to five. This parameter, however, doesn't differentiate between work order types; all are treated equally. The %cfg_reservations hash allows one or more job types to reserve portions of the available job slots just for themselves.

If $cfg_max_children is greater than the total number of reservations so allocated, "walk-ons" will be accepted into any extra slots.

For example:

```
$cfg_max_children = 8;
%cfg_reservations = (
    'ALLOCATE_MoPGE01' => 1,
    'ALLOCATE_MoPGE02' => 5,
);
```

In the example above, a maximum of eight jobs can be running at the same time. Five of the six are reserved for job types of ALLOCATE_MoPGE02 and one is reserved for ALLOCATE_MoPGE01. Since the $cfg_max_children is set to more than the total, two slots are for walk-ons of either job type.

Note that if some job types do not have reservations specified, they can **only** be accepted as walk-ons. Thus in this example, if a ALLOCATE_MoPGE55 work order showed up (assuming the station was otherwise configured for it), it would only run when extra slots became available.

Be sure to set the $cfg_max_children so that it is equal to or greater than the total number of reserved job slots.

Note also that using reservations is functionally equivalent to creating separate S4P stations, one for each job type needing a reserved number of job slots.

4.5 Common Optional GUI Parameters

Station isn't the only part of S4P that looks at the station configuration files for determining what to do and how to behave. The S4P Monitor (aka tkstat.pl and tkjob.pl) also uses the same station configuration file to override defaults and provide options. This section concentrates on those parameters in the station configuration file that affect the S4P Monitor.

Parameter	Description	Section
$cfg_station_name	Sets station name on button in S4P Monitor	4.5.1
%cfg_max_job_time	Sets time beyond which a job box will change from green to yellow	4.5.2
%cfg_failure_handlers	Associates job failure handlers with the station	4.5.3
%cfg_manual_overrides	Associates commands to run on running jobs	4.5.4
%cfg_interfaces	Associates station-level commands or GUIs with the station	4.5.5

Table 4-3. Common station configuration file GUI parameters.

4.5.1 Station Name ($cfg_station_name)

The $cfg_station_name parameter sets the name of the station as it is to appear in the S4P Monitor (tkstat.pl). It is not used by Station. For very small S4P deployments where the S4P Monitor will not be used, this parameter is unnecessary. By default, the S4P Monitor will display the station directory name on the Station buttons down the left side of the interface. If $cfg_station_name is set, it will use the value set here.

For example:

```
$cfg_station_name = 'Import Data';
```

By tradition, station directory names are all in lowercase with underscore characters separating words and their station name counterparts set with the $cfg_station_name parameter are the same, but with mixed case and separated by spaces. Thus, the station directory name corresponding to the above example would be: import_data. This is, however, not enforced by S4P.

4.5.2 Maximum Job Time By Work Order Type (%cfg_max_jobtime)

The %cfg_max_jobtime parameter is a hash that allows one to preset the maximum number of seconds (wall clock time) that a job processing a work order of a particular type should run. This hash is used only by the S4P Monitor (tkstat.pl) and not by Station. When a running job exceeds the time specified here, the monitor changes the color of the job box from green to yellow. A yellow job is still a running job. The purpose of this option is to allow those monitoring jobs in a S4P string to recognize when a job is taking too long and may need to be investigated.

For example:

```
$cfg_max_jobtime = {
        'FIND_ALG1' => 100,
        'FIND_ALG2' => 100,
};
```

In the above example, jobs with work orders of type FIND_ALG1 and FIND_ALG2 are given a maximum time of 100 seconds each. The implication is that jobs taking longer than this to run are in trouble.

4.5.3 Failure Handlers (%cfg_failure_handlers)

Jobs in S4P sometimes fail by exiting with a non-zero exit code. As discussed earlier, when a job fails, Station changes the name of the job's subdirectory from RUNNING.<jobtype>.<jobid> to FAILED.<jobtype>.<jobid>. This makes it easy for one to see the failures and, hopefully, resolve the problems causing them.

In the S4P Monitor, a failed job is represented by a red box.

Just as clicking on a green running job box brings up the S4P Job Monitor (tkjob.pl) in that running subdirectory, clicking on a red job box brings up the S4P Job Monitor in the failed subdirectory. S4P can be configured to present the operator with actions to take when a job fails. These actions are only offered to the operator when the job has already failed. The GUI will not offer the action in a running job directory (there is another option to handle that).

Actions that can be invoked on a failed job are configured for a station via the %cfg_failure_handlers hash as in:

```
%cfg_failure_handlers = (
```

```
        'Remove Job' => 'remove_job.pl',
        'Restart' => 'restart_job.pl && remove_job.pl'
);
```

The hash keys are button labels that appear on in the S4P Job Monitor interface and the hash values are those things to run when the action is invoked.

In the above example, when a job fails an operator will be presented with two options via the S4P Job Monitor. One is to remove the job and the other is to restart it (presumably after some fix was applied) and remove the failed one at the same time.

S4P comes with the remove_job.pl and restart_job.pl scripts. Other scripts or commands can be easily added.

When developing custom made failure handlers, there are some things to keep in mind:

- The current working directory is the failed job directory. The failure handler needs to assume this.
- It is completely up to the failure handler itself to move work orders (if any) to other stations since Station isn't running. When moving work orders, remember the required naming convention that needs to be observed.
- It is also up to the failure handler itself to move the chain log to other stations. Normally, Station handles this automatically, but again, it's not running here.
- Failure handles are free to examine any debris left by the failure to decide what to do. This includes the input work order (which should still be in the directory) as well as log and other files.

Note that failure handlers defined as such can be run without the S4P Monitor or S4P Job Monitor by simply changing directories into the failed job directory and running the handler on the command line.

4.5.4 Station Manual Overrides (%cfg_manual_overrides)

The %cfg_failure_handlers hash described above allows operator actions to be taken on jobs that failed in S4P. What if you want to take some action on a job that is still running? That is the purpose of the %cfg_manual_overrides hash. It allows something to be run in a jobs running directory while the job is still running. The hash keys are labels for buttons that will appear in the S4P Job Monitor interface and the hash values are the commands to run. For example:

```
%cfg_manual_overrides = (
        'Expire Current Timer' => 'touch EXPIRE_CURRENT_TIMER',
        'Ignore Optional' => 'touch IGNORE_OPTIONAL',
        'Ignore Required' => 'touch IGNORE_REQUIRED'
);
```

In the above example, each of the actions taken is the creation of an empty file (via the UNIX 'touch' command) in the job's running subdirectory. In this case, we presume that

the job that is running polls periodically for these "signal" files and takes some action depending upon which signal file it sees. This is just one example of the utility of %cfg_manual_overrides.

As with failure handlers, keep in mind that the current directory of any manual override you design is the running job directory.

4.5.5 Station Interfaces (%cfg_interfaces)

By default, right clicking on a station button in the S4P Monitor displays options for starting or stopping the station. These same options are also visible via radio buttons in the interface that comes up when you just click on a station button.

Through the station configuration file, other options can be added on a station-by-station basis via the %cfg_interfaces hash. For example:

```
%cfg_interfaces = (
        'Restart All Failed Jobs' => 's4p_restart_all_jobs.pl'
);
```

In this example, the station is configured with an option to restart all failed jobs in the station. This option shows up in addition to the start or stop option.

The s4p_restart_all_jobs.pl is included in S4P and it does exactly what it says: it simply executes a restart_job.pl and remove_job.pl for all failures. This can be useful when there is a simple glitch that causes many jobs to fail. Once fixed, rather than restarting each job individually (by clicking on the red box and then on Restart Job), this item can be selected from by right clicking on the station button.

Custom interfaces can be added easily. Unlike with failure handlers, custom interfaces are run from within the station directory, not in any one job subdirectory. Keep this in mind when designing your interface.

4.6 Useful, Yet Less Common Options

This section will discuss some of the less common configuration parameters that can be set in the station configuration file.

Parameter	Description	Section
$cfg_ignore_empty_files	Don't fail if work order is empty	4.6.1
$cfg_ignore_duplicates	Don't fail if a work order is duplicated	4.6.2
$cfg_case_log	Log certain messages in a case-based reasoning log file	4.6.3
$cfg_max_failures	Shut station down after a maximum number of failures	4.6.4
%cfg_token	Execute job only via tokens from a token master station	4.6.5
%cfg_auto_restart	Automatic restart of failed jobs	4.6.6

Table 4-4. Useful, yet less common station configuration file optional parameters.

4.6.1 Ignore Empty Work Orders ($cfg_ignore_empty_files)

This option tells Station to ignore recognized work orders that are zero bytes long. Note, use this carefully since stations can legitimately be configured to use empty work orders as "signal" files, that is, the work order triggers processing, but that processing doesn't require any information from the work order.

This parameter may be useful on systems with overloaded file systems that result in work order poaching (see Section 4.4.2). Rather than causing failures, the empty work orders are quietly ignored (they will be noted in the station's log file, by default station.log).

Set to non-zero to enable; set to zero or leave out to disable. For example:

```
$cfg_ignore_empty_files = 1;
```

4.6.2 Ignore Duplicate Work Orders ($cfg_ignore_duplicates)

This parameter tells Station to ignore cases where duplicate work orders have apparently been sent to the station. This can happen when file system response exceeds the station polling interval ($cfg_polling_interval).

4.6.3 Case-Based Reasoning Log ($cfg_case_log)

The $cfg_case_log parameter enables or disables the logging of messages that can be used to build case-based reasoning into a S4P. To enable, set to the location and file name of the log file into which these messages will be written. Leave out or set to undef to disable case-based reasoning logging. If multiple stations will be logging, a common log file in the S4P root directory is recommended; all station configuration files can refer to this single log file. For example:

```
$cfg_case_log = '/s4p/mystring/stations/cbr.log';
```

To find out information on what case based reasoning is, go to your favorite Web browser and do a search. There is a lot of information out there, more than can be given justice here.

Each entry in the case log file consists of up to seven items:

1 Timestamp – A time stamp, number of seconds since January 1, 1970
2 Message Type – 'F' for failure, 'M' for a manual intervention (*e.g.* killing a job, restarting a job, or other manual override set in %cfg_manual_overrides)
3 Exit Code – Exit code from the failure (if applicable)
4 Station Name – Station involved (if applicable)
5 Job Type – Work order job type
6 Job ID – Work order job ID
7 Information – Additional information

As an example, here are a few lines from such a log file:

```
1154454319 F   9 run_algorithm RUN_MoPGE02  "Job Failed"
1154454327 R   0 run_algorithm RUN_MoPGE02  "remove_job.pl "
1154454334 R   0 run_algorithm RUN_MoPGE02  "remove_job.pl "
1154527983 M   0 reprocessing   "Execution of [s4pshutdown.pl -r]
succeeded"
1154830881 M   0 find_data FIND  "Execution of [touch EXPIRE_TIMER]
succeeded "
```

In order to take advantage of this logging, the scripts running within the S4P station must invoke the S4P::log_case() function and provide it the information that is to be logged. Since this function is in Perl, it can only easily be used by scripts written in Perl.

4.6.4 Maximum Failures ($cfg_max_failures)

The $cfg_max_failures sets the maximum number of failed jobs that can pile up in a station before the station shuts down. It looks at the number of job subdirectories that start with FAILED. Thus, failed work orders (work order types not recognized) that are in the FAILED.WORK_ORDERS directory are not counted here.

The default is to allow an infinite number of failed jobs.

For example:

```
$cfg_max_failures = 20;
```

To limit maximum failures to particular job types while placing no limits on others within a station:

```
$cfg_max_failures = {
    'SELECT' => 5,
    'SELECT_ALL' => 4,
};
```

4.6.5 Use Tokens (%cfg_token)

The %cfg_token parameter is a hash whose hash keys are job types and hash values are the directories of the token master station responsible for providing run tokens to this station. When used, a job will be spawned but will not begin executing until a token is received.
Tokens are used in S4PM to allow a number of stations (within a string or among several strings) to share CPU resources. For example, to have the Run Algorithm stations of two S4PM strings on the same machine share CPU resources, you could set up a token master station configured with the maximum number of tokens equal to the number of CPUs on the machine. The token master station sends available tokens (a work order) to one of the Run Algorithm stations. That Run Algorithm station executes a spawned, yet pending job and then release the token back to the token master after the job is complete.

The token master station itself it not part of any one S4PM string. The script for the token master station is s4p_token.pl.

4.6.6 %cfg_auto_restart

The %cfg_auto_restart can be used to automatically restart failed jobs of a given job type. Station will first run a scout job to see if the "coast is clear." If successful, all failed jobs of the specified job type will automatically be restarted. This feature can be useful in situations where problems may go away on their own without intervention (e.g., network congestion or remote server being down). The hash keys are the job types and the hash value is itself a hash containing (for now) one element, the restart interval (in seconds). This interval specifies how long to wait before attempting to restart jobs by running a scout job; it should be relatively long.

For example:

```
%cfg_auto_restart = (
    'FTP_CONNECT' => {'interval' => 3600},
    'FTP_PUSH' => {'interval' => 86400},
);
```

4.7 Obscure Options

4.7.1 Failed Work Order Directory ($cfg_failed_work_order_dir)

By default, work orders whose types are not recognized end up in the FAILED.WORK_ORDERS subdirectory (under the station directory). To change the name to something else, use $cfg_failed_work_order_dir as in:

```
$cfg_failed_work_order_dir = 'MISTAKES';
```

4.7.2 Station Log File Name ($cfg_logfile)

By default, log file for Station is station.log. To change it to something else, use $cfg_logfile as in:

```
$cfg_logfile = 'messages.log';
```

4.7.3 Station Counter Log File Name ($cfg_counter_logfile)

By default, Station logs information in a file named station_counter.log. This information allows the S4P Monitor to keep track of the number of jobs that succeeded and the number that failed since some point in time. To change the name of this file, use $cfg_counter_logfile as in:

```
$cfg_counter_logfile = 'counter.log';
```

4.7.4 Child Process Sleep ($cfg_child_sleep)

When Station running in a station detects a work order to process, it forks off another Station child process to monitor the execution of that work order. As a default, the Station parent process sleeps two seconds after the job subdirectory is created and the work order is moved into that directory before starting up the child Station process. To change that default to something else, use $cfg_child_sleep.

Note, if $cfg_max_children is set to zero, no child Station process is started up in the first place and therefore, this parameter has no impact.

```
$cfg_child_sleep = 3;
```

4.7.5 Rename Retries and Retry Interval ($cfg_rename_retries, $cfg_rename_retry_interval)

The $cfg_rename_retries and $cfg_rename_retry_interval are parameters that allow Station to make multiple attempts to renaming a RUNNING job subdirectory into a FAILED job subdirectory. These are intended to get around cases where the rename fails, but may succeed in a little while, such as when Windows holds onto file in the directory a little too long.

The default for $cfg_rename_retries is zero (it only tries once) and the default for $cfg_rename_retry_interval is five seconds. For example:

```
$cfg_rename_retries = 2;
$cfg_rename_retry_interval = 5;
```

In the above example, there will be two retries separated by 5 seconds in time.

4.7.6 Virtual Jobs (%cfg_virtual_jobs, $cfg_virtual_feedback)

The %cfg_virtual_jobs hash is used to create virtual work orders. Virtual work orders are work orders created by Station itself rather than by another station or process. Such work orders contain no content. They are merely used to initiate some processing and they are recreated according to a set interval.

For %cfg_virtual_jobs, the hash values are the job types to create virtually and the hash values are the number of jobs to maintain (as either running or in the queue to be run). As an example:

```
%cfg_virtual_jobs = (
    'FOOBAR' => 2,
);
```

The above example will cause Station to maintain two jobs of type FOOBAR.

If $cfg_virtual_feedback is set, failed jobs will be counted along with running and queued jobs for Station's reckoning of jobs to maintain. For example:

```
$cfg_virtual_feedback = 1;
```

Using the %cfg_virtual_jobs, Station will create work orders named according to DO.<jobtype>.<jobid>.wo where the <jobtype> is specified as the hash keys.

With this option, the station script should not generate output work orders as Station will be doing this instead.

5. S4P Graphical User Interfaces

For simple S4P installations with few stations processing a relatively few jobs, monitoring can be done via a simple examination of directories, subdirectories, and file. For larger system processing many jobs, this may be impractical. For this reason, a simple graphical user interface (`tkstat.pl`, written in Perl/Tk) is provided in S4P for monitoring. It simply displays the number of running and failed jobs in a station as well as whether a station is currently "up" (has a Station daemon running) or "down" (has no Station daemon running). An additional program, `tkjob.pl`, is invoked by `tkstat.pl` to drill down to the station or job level, allowing the operator to examine configuration and log files, or even to start or stop stations.

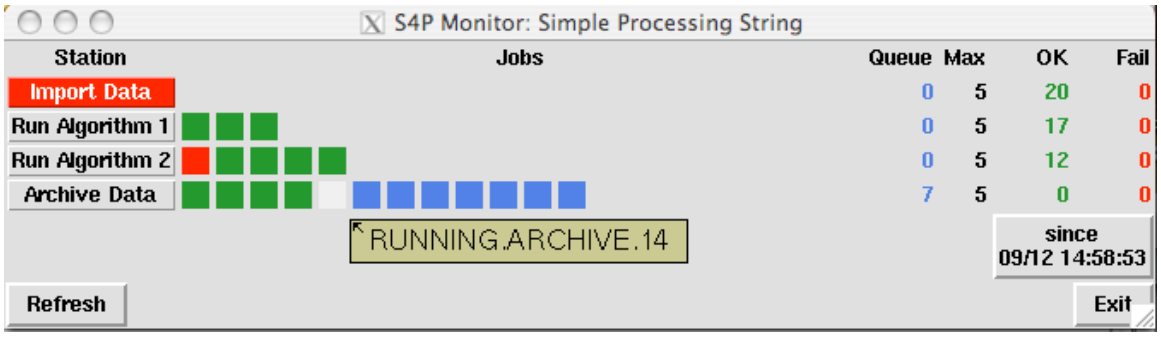

Figure 3-1. Simple graphical user interface, tkstat.pl, used in S4P. This one displays four stations (used as an example in a later discussion).

5.1 Anatomy Of The S4P Monitor (tkstat.pl)

The S4P Monitor is the main interface for monitoring activity and status of a S4P string. The S4P Monitor updates the display on a configurable frequency. The default is ten seconds. Figure 1-1 shows an example S4P Monitor implemented in the tkstat.pl Perl script.

To start up the monitor as displayed in Figure 3-1, the following command was run from the directory below which reside the four station subdirectories (should be on a single line):

```
tkstat.pl -t 'S4P Monitor: Simple Processing String' import_data
run_algorithm_1 run_algorithm_2 archive_data &
```

In the sections below, each of the aspects of the S4P Monitor is discussed:

5.1.1 Title Bar

The title bar at the top of the window is labeled S4P Monitor and also displays a name that can be assigned on the command line when invoking tkstat.pl via the –t option.

5.1.2 Station Buttons

The station buttons run down the left side of the window and represent stations configured to work in this S4P string. If the station button is red (Import Data, in this example), it means that station is not running. If the station button is yellow (not shown), it means that there is an anomaly in that station, but that the station is still running. Although not shown in Figure 2-1, the S4P Monitor can be configured to display *all* stations, even those not active for the current configuration. Such stations would be shown as red, but with gray text to distinguish them from active stations that are simply not running at the moment.

To have only active stations displayed in the S4P Monitor, include the –F option on the command line to tkstat.pl.

Clicking on a station button brings up the Station Monitor window that will be discussed below.

Right clicking on a station button is a shortcut to some of the things you can do in the tkjob.pl Station Monitor such as stopping or starting the individual station. When you right-click on a station button, a popup menu allows you to select these options as well as any others configured for that station.

5.1.2.1 Station Anomalies

Station anomalies are caused by problems that are not severe enough for the station to be turned off yet they do warrant investigation. Such stations turn yellow. This behavior is triggered by any file detected in the station directory with the name:

ANOMALY-xxxx.log

where xxxx can be any string (any case, although upper case is shown here). For example:

ANOMALY-BAD_FS.log

indicating a bad file system will cause the station button to turn yellow. The content of anomaly files is up to the application creating them. By default, S4P supports several built-in anomalies:

Built-In Anomaly	Anomaly File Name	Description
MAXFAIL	ANOMALY-MAXFAIL.log	Triggered when $cfg_max_failures is exceeded.
SLOW_FS	ANOMALY-SLOW_FS.log	File system is slow. Detected by unexpected zero-length work orders or duplicate work orders
BAD_CONFIG	ANOMALY-BAD_CONFIG.log	An unrecognized work order type was detected, one that ended up in the FAILED.WORK_ORDERS subdirectory.
BAD_PERM	ANOMALY-BAD_PERM.log	Bad permissions as recognized by failure to write a virtual job work order.

Station scripts can add other anomalies freely simply by following the above file name convention. Writing content to the anomaly file is optional (though perhaps useful). To facilitate anomaly handling in station scripts, two functions are available: S4P::raise_anomaly() and S4P::clear_anomaly(). For more information on how to use these functions, refer to the documentation in S4P.pm.

To clear an anomaly (presumably after it has been investigated), the operator need only select the anomaly file and then click on the "Clear Anomaly" button in the Station Monitor. This causes the selected anomaly file to be deleted. The station will stay yellow until all anomaly files have been cleared in this manner.

5.1.3 Job Boxes

Jobs running in each of the stations are represented with colored boxes to the right of the station button. Green boxes are running jobs. Yellow job boxes (none shown here) also are also running jobs but ones that have been running more than a configurable amount of time; such jobs may be an indication of a problem worth investigating. Jobs waiting to be run (in the queue) are represented with blue boxes. Jobs that have been suspended (and waiting to be resumed, presumably) are represented in dark blue. Finally, jobs that have failed are represented with red boxes.

Hovering over a job box with the mouse displays the job identification, as is illustrated in Figure 2-1 for a running job in the Archive Data station (hovering temporarily obscures the color of the box).

In Figure 2-1, the Import station is turned off and there are no jobs at all in the station. The Run Algorithm 1 station has three running jobs (green). The Run Algorithm 2 station has one failed job and four running jobs. The Archive Data station has five running jobs (green) and seven queued jobs (blue). In addition, the mouse cursor is hovering over the last of the running jobs and a bubble displays the name of the running job.

Each station is configured for the maximum number of jobs that can be run at the same time. The default is five. Additional jobs that come into the station after the maximum is reached are queued up and are displayed as blue boxes. In Figure 2-1, the Archive Data station is configured to only run the default of five jobs at a time. Thus, there are five green boxes and the remaining jobs are in the queue. Once one of the running jobs finishes or fails, one of the queued up blue jobs will then be run.

When a running job (green box) completes successfully, it disappears. If it fails, the box turns red and remains there until some action is taken. A red box indicates that something is wrong and should be tended to.

Jobs that have been suspended are shown in midnight blue (a darker blue than is used for queued jobs). None are shown in this example.

5.1.4 Queued, Max, OK, and Fail Tallies

On the right side of the window are four columns that maintain current tallies of jobs that are in the queue, the maximum number of jobs configured to run in the station, the

number of jobs that have completed successfully (OK), and the number of jobs that have failed since the date and time listed on the **since** button. Clicking the **since** button resets all tallies back to zero.

5.1.5 Control Buttons

At the bottom of the window are a number of S4P control buttons placed here for convenience. By default, there is only a Refresh button and an Exit button. More buttons carrying out useful functions, however, can easily be added via a configuration file indicated on the tkstat.pl command line via the –c option.

5.1.6 Adding Custom Control Buttons

As mentioned in the previous section, custom control buttons can be added to the S4P Monitor via the –c option when starting up tkstat.pl. The –c option takes the name of an optional configuration file as its argument. This configuration file contains the single hash %tkstat_commands where the hash keys are button labels and the hash values are the associated actions that are to be invoked by clicking on the button. The maximum number of user-defined buttons is six (in addition to the standard Exit and Refresh buttons).

By default, the order of buttons as displayed in the S4P Monitor is random, but it can be specified explicitly by prepending a number followed by a colon to the button names (see example below). The number and colon will be stripped off when displayed. The order of the buttons will be in number order. Note that the Refresh button is always displayed on the far left and the Exit button on the far right; user-defined buttons are displayed in between these.

An additional feature is that any button label containing the string “kill” will be displayed with red text.

As an example, the hash below defines two buttons to start up and shutdown all stations. The order will be random, but likely to be in the order shown. The new buttons will be labeled on the display with the hash values shown and will be situated between the Refresh and Exit buttons.

```
%tkstat_commands = (
        'Start All => "cd /s4pm/test/stations; s4p_start.sh",
        'Stop All' => "cd /s4pm/test/stations; s4pshutdown.pl",
);
```

To guarantee that the Start All button preceeds the Stop All button, you can do this:

```
%tkstat_commands = (
        '1:Start All => "cd /s4pm/test/stations; s4p_start.sh",
        '2:Stop All' => "cd /s4pm/test/stations; s4pshutdown.pl",
);
```

Note that s4p_start.sh replaces the deprecated s4pstart.ksh script.

If the file with the above hash is saved as tkstat.cfg, then it can be applied by running this modified command to start up tkstat.pl:

```
tkstat.pl -t 'S4P Monitor: Simple Processing String' -c tkstat.cfg
import_data run_algorithm_1 run_algorithm_2 archive_data &
```

5.1.7 Command Line Options

The table below contains the full list of options for tkstat.pl, the S4P Monitor.

Option	Description
-a *appName*	Sets the application name as recognized by the X resources database. This allows you to start it up with different X resource values.
-f *font*	Sets the font for the interface; must be quoted if the font name begins with a hyphen (as most do).
-m *max_blocks*	Number of blocks or cells to display in each row of the display. The default is 15.
-r *refresh_rate*	Interface refresh rate. The default is 5 seconds.
-b	Use bitmaps to display the job status. If this option is chosen, the following default bitmaps are used: FAILED –circle with a diagonal line through it WARNING – exclamation mark RUNNING – hourglass PENDING – question mark
-t *title*	Sets the title to display in the title bar of the interface
-c *config_file*	Full pathname of the configuration file for adding in custom buttons. See Section 5.1.6.
-R	Make interface read only (e.g. for managers).
-C	Run in “classic” mode meaning that it will run the deprecated stationmaster.pl rather than the newer s4p_station.pl. At some point, this option will likely be dropped.

Table 5-1. List of all tkstat.pl options and their descriptions.

5.1.8 X Resources

In addition to command line options, tkstat.pl also supports X resources from the useer’s .Xdefaults or .Xresources file. Aside from the usual ones, the following are supported as well:

Resource	Description
failedColor	Sets the color for display of failed jobs. Default is red (class FailedColor)
pendingColor	Sets the color for display of pending jobs. Default is RoyalBlue (class PendingColor)
runningColor	Sets the color for display of running jobs. Default is forest green or #228b22 (class RunningColor)
warningColor	Sets the color for display of late-running jobs. Default is yellow (class WarningColor)
suspendedColor	Sets the color for display of suspended jobs. Default is MidnightBlue (class SuspendedColor)
failedBitmap	Sets bitmap for failed jobs if –b option is set. Default is circle with a line through it (class FailedBitmap)
pendingBitmap	Sets bitmap for pending jobs if –b option is set. Default is a question mark (class PendingBitmap)
runningBitmap	Sets bitmap for running jobs if –b option is set. Default is an hourglass (class RunningBitmap)
warningBitmap	Sets bitmap for late-running jobs if –b option is set. Default is an exclamation mark (class WarningBitmap)
maxBlocks	Set the maximum number of jobs to display in a row. The default is 15 (class MaxBlocks)

Table 5-2. List of X resources supported by tkstat.pl and their descriptions.

5.2 Controlling Stations With Station Monitor

Clicking on a station button in the S4P Monitor brings up another window. An example for an imaginary Allocate Disk station is shown below:

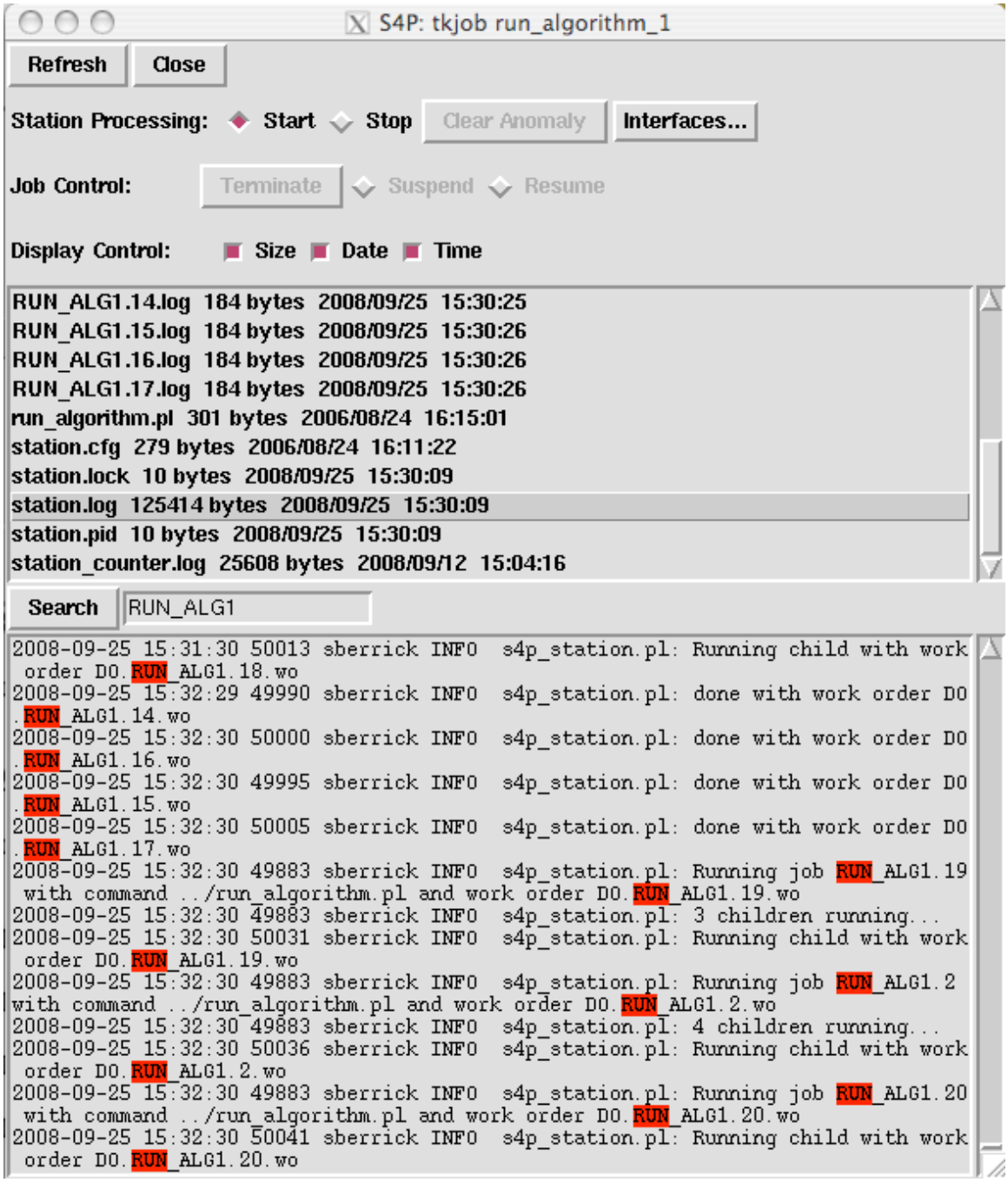

Figure 5-2. Example S4P Station Monitor, in this case, for the run_algorithm_1 station.

The S4P Station Monitor is basically a view into the station's directory (remember, all stations are just directories): the top panel has some buttons and controls, the middle panel is a listing of the station directory's contents, and the bottom panel shows the contents of the file selected from the middle panel. In this example, the station is Run Algorithm 1 selected from our previous example. and the actual station directory's name, run_algorithm_1, is shown in the title bar of the window. The contents of the file highlighted in the middle panel, station.log, are shown in the bottom panel. In addition, the search mechanism is also illustrated.

5.2.1 Anatomy Of The Station Monitor

The Station Monitor is the interface to a particular station running in S4P. As with the main S4P Monitor, the Station Monitor is updated periodically.

Below, each of the aspects of the Station Monitor is discussed:

5.2.1.1 Top Panel

The top panel of the Station Monitor contains buttons for controlling that station. These are common to all S4P stations. These are:

Button Name	Function
Refresh	To update the display now rather than waiting for the next polling cycle (default is 5 seconds)
Close	To close the Station Monitor window
Start and Stop Radio Buttons	To start or stop the station
Terminate	To kill a running job. This button is only active when a job is running; inactive otherwise.
Suspend and Resume Radio Buttons	To suspend and resume a job. Suspend is only available when a job is running and has a function similar to Control-Z in UNIX. Suspended jobs are represented as dark blue in the S4PM Monitor.

Table 5-3. Station controls available in the Station Monitor window that are common across all S4PM stations.

In addition to these controls, a number of station-specific control buttons may be configured to carry out tasks unique to those stations. We've seen how to do this via the station configuration file and the use of the %cfg_interfaces hash.

Note that the functionality of the **Start** and **Stop** Radio Buttons as well as any station-specific interfaces is also available by right-clicking on the station button in the S4P Monitor window. Thus, you don't have to bring up a new window to get access to these controls. Any unique interfaces added to a station would be available in the same way.

Finally, the **Display Control** check boxes control the display of directory contents in the middle panel.

5.2.1.2 Middle Panel

The middle panel is a display of the contents in the station directory including other directories. Through this panel, you can navigate to other directories by double-clicking on the name. For example, double-clicking on .. (the two dots) will bring you up one directory.

A single click on a file will result in the contents of that file getting displayed in the bottom panel. Only ASCII text files can be displayed along with Perl DBM files (typically having a file name extension of .db). A single click on a directory will result in

the contents of that directory getting displayed in the bottom panel. In this example, the user has clicked on the file station.log which is highlighted. The contents of that file are displayed in the bottom panel.

5.2.1.3 Bottom Panel

The bottom panel is used for displaying the contents of the file or directory highlighted in the middle panel. If nothing is selected in the middle panel, the bottom panel will be blank.

To help with diagnosing problems, any file name having the file name extension .log is assumed to be a log file and the display of that file will automatically go to the bottom where, most typically, the proximate cause of any problem can be found.

To facilitate troubleshooting, a rudimentary search button and entry box is provided. All matches will be highlighted in color in the file shown in the bottom panel and the display will jump to the first occurrence found. In this example, a search was done on the string "RUN_ALG1".

5.3 Controlling Jobs With Job Monitor

Clicking on a running job (green box) or a failed job (red box) brings up the Job Monitor window. The Job Monitor is essentially the same as the Station monitor, except that rather than showing you the contents of a station directory, the Job Monitor shows you the contents of a job directory (again, everything's just a directory). If the job is green and still running, the contents of the directory are likely to be continuously changing. If the job has failed, the contents are instead the debris left behind by the job's failure.

5.4 Relationship Between Station Monitor and Job Monitor

As indicated above, the Station Monitor and Job Monitor are essentially the same window. The only distinction is the directory being viewed. If the directory is a station directory, the interface is a Station Monitor; if the directory is a job directory within a station directory, the interface is a Job Monitor. You can verify this yourself by bringing up the Job Monitor on some running or failed job, then navigating in the middle panel to one directory up (double click on the ..) into the station directory. Viola! The interface will change from being a Job Monitor to a Station Monitor. The control buttons in the top panel will even change appropriately. Furthermore, you can navigate back down to another station directory. Again, the control buttons that show up will be those for that new station (some may disappear, others may show up anew).

6. Application to Automated Science Processing

Many science processing systems are data-driven, in the sense that incoming data triggers the processing to be done. An example of this is the classic AVHRR 1 km system at EDC. The alternative paradigm is that of planned processing, where plans of the processing to be done are constructed before data arrival. The EOSDIS Core System (ECS) exemplifies this approach. S4P is an example of the data-driven type (and is indeed loosely based on the AVHRR system in concept.)

6.1 Simple Processing

Let us assume that an external data source is pushing data files to us. We will then process each data file through an algorithm that produces a single output data file for each input. We then process the output data file through a second algorithm that produces a single output file for each input, and finish by archiving the second output. Such a system might be implemented with only 4 stations (we've actually used this example before):

Station	What The Station Does
Import Data	Detects the arrival of input data and passes the full pathnames on to the Run Algorithm 1 station.
Run Algorithm 1	Runs the first algorithm, deletes the input data, and passes the full pathname of the output data on to Run Algorithm 2 station.
Run Algorithm 2	Runs the second algorithm, deletes the input data, and passes the full pathname of the output data to the Archive Data station.
Archive Data	Saves the data to an archive.

Table 6-1. Four S4P stations that implement a simple processing scenario.

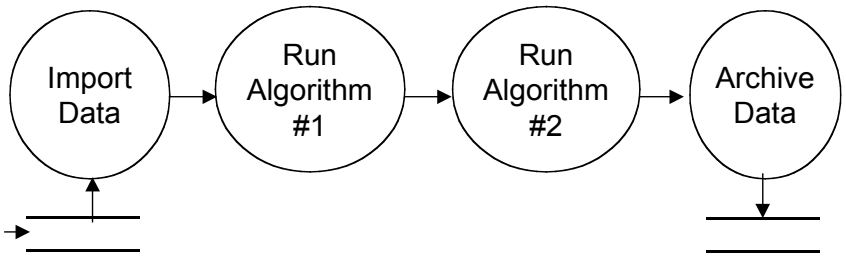

Figure 6-1. Simple processing scenario.

This system assumes very simple production rules (*i.e.,* one file in, one file out), and no resource contention (*i.e.,* more than enough disk space to go around. Figure 3-1 showed how this simple scenario would be monitored via the S4P Monitor.

6.2 Dealing with Complicated Production Rules

In practice, production rules can be quite complicated: ancillary files may be optional or required; input files may be processed in groups; adjacent files may be needed for averaging, and so on. In fact, the variety of production rules is potentially boundless. As a result, such rules are not supplied as an intrinsic part of S4P (they are, however, an intrinsic part of S4PM). Instead, they are implemented as stand-alone stations, sometimes called triggering stations (Figure 6-2). In this case, the triggering station collects information about all of the input files needed and passes that information to the algorithm run station. As such, the program run in the triggering station can be arbitrarily complicated and written in any language. It needs to fill only two criteria:

1. It is a standalone executable program (*i.e.,* binary or script).
2. It outputs a file intelligible to the corresponding Run station.

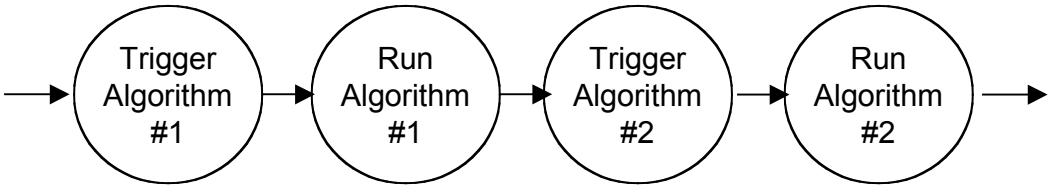

Figure 6-2. Triggering stations.

Of course, the logic in the Trigger station could be prepended to that in the corresponding Run station, thus halving the number of stations needed for the purpose. However, there are two advantages to a separate Trigger station when the rules are complicated:

1. The complicated triggering logic is separated from the algorithm, producing a more modular, maintainable system. Indeed, it can even be developed separately from the main algorithm stations.
2. The triggering can be monitored separately from the algorithm execution, which is helpful because the types of errors occurring in each one tend to be quite different.

The downside of the above approach in S4P is that if the production rules change for an algorithm, the code implementing the trigger station will need to be modified or perhaps rewritten. For simple processing situations where algorithms are relatively stable, this is still a very cost effect approach.

On the other hand, S4P is the basis for S4PM which supports in a generic and rich set of production rules along with data management and tracking. For more information on S4PM, see http://s4pm.sci.gsfc.nasa.gov/.

6.3 File Tracking

Most complicated production rules tend to deal with the use of multiple files, sometimes of different data types, but also occasionally groups of files of the same data type. For

instance, the MODIS Level 1B (Calibration) algorithm processes an input Level 1A file using the adjacent (leading and trailing) Level 1A files to average the calibration information, as well as the corresponding Geolocation file. For such cases, it is also important to track what files are available and whether they have been used for all of their intended purposes.

File tracking can be implemented in a number of ways. The first main category is the use of a database, either relational, DBM or otherwise. In this case, stations that make or bring in data populate the database, while Triggering stations query it for available data (Figure 3-3.)

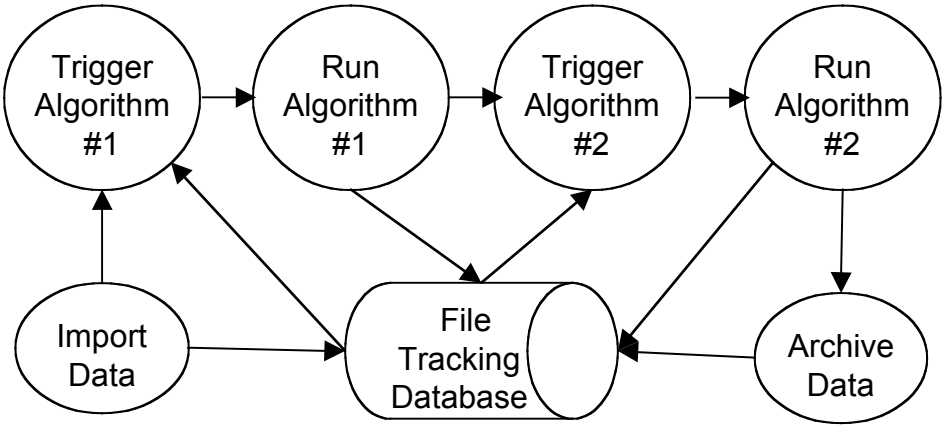

Figure 6-3. Database implementation of file tracking.

Alternatively, a station can be used to track the files in the system. In this case, all file creations or updates generate work orders which are sent to the File Tracking station. "Queries" are accomplished by sending work orders with placeholders to the this station, which fills them in with the necessary file pathnames.

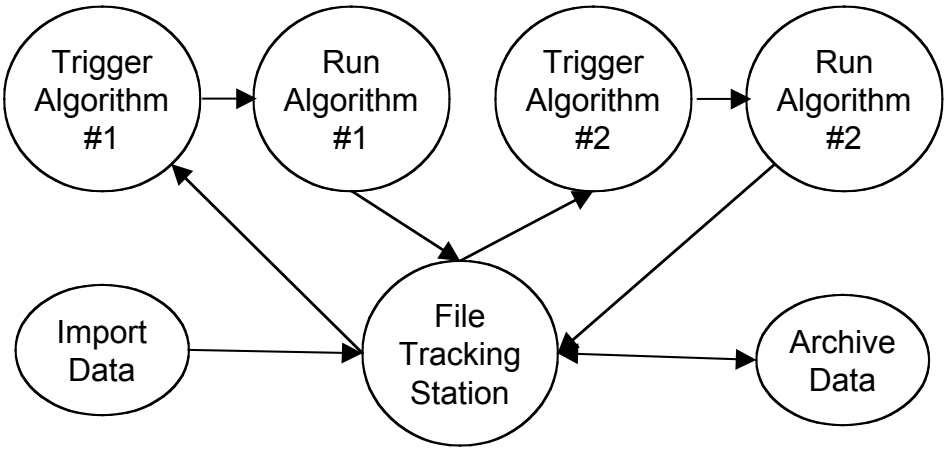

Figure 6-3. Station implementation of file tracking.

7. Getting Started

7.1 Create your S4P directory structure

Just use the `mkdir` command in either Unix or Windows to do this. It's a good idea (though not essential) to create all of the directories in a subdirectory of their own. Don't worry about the space used by these; the data that are processed need not be on the same disk, though it should be accessible from this area.

In designing your directory structure, you will need to make decisions about "lumping" tasks vs. "splitting" tasks. It can be simpler (especially in the beginning) to lump tasks together into a small number of stations (*e.g.,* by having a script run them in sequence). On the other hand, this limits your ability to see what is going on within the "lump", as the monitoring granularity is at the station level. However, it makes sense to lump two tasks when one of those is a highly routine operation that is unlikely to break down.

If you have a Data Flow Diagram (DFD) or Object diagram describing your processing system, it often makes sense to have one station per process or object.

7.2 Create your station configuration files

The station configuration file lives in the station directory. The default name for the file is `station.cfg`, although the Station can be brought up with different file names using the command line arguments (`man s4p_station.pl` for more details).

7.2.1 Work Order / Executable Map

Typically, a station runs only one thing. But it may run several. For each thing you want to run in a station, come up with a work order type. The type can be arbitrary, but it should be something that conveys what type of processing is being done. Thus, in our earlier example, the work order type IMPORT makes sense for an import station. Likewise, the work order type RUN_ALG1 makes sense for the Run Algorithm 1 station. The decisions you make here determine how to specify the %cfg_commands hash in the station configuration file.

7.2.2 Downstream Work Order Map

In general, a station generates output work orders destined for downstream stations. So, the next step is to decide the work order types of those output work orders. You should give them names that convey the type of processing that will be done in those downstream stations and, of course, they need to match what those downstream stations are configured to accept. This determines how to specify the %cfg_downstream hash in the station configuration file.

7.2.3 Other Configuration Parameters

Station configuration file parameters were discussed in Section 4. Now is the time to decide, for each station, what of those parameters make sense. Typically, if you plan on using the S4P Monitor, a nice station name is desirable, that is, $cfg_station_name. Also, to simply configuration, the $cfg_root parameter is useful.

Here's an example from the S4P system processing MODIS Direct Broadcast data:

```
$cfg_station_name = "Run L1A";
$cfg_root = '/usr/modis/db/s4p/';
%cfg_commands = ('MOD00' => 'perl ../run_pge.pl -l 150 -a 1 -o
/usr/modis/db/data/1');
%cfg_downstream = ('MOD01' => ['mod_pr03']);
```

7.3 Start up Station daemons.

You can do this manually, by running s4p_station.pl (from the command line or in a script), or using the tkstat.pl monitor, by clicking the button for a station (a red background means it is not currently running), and clicking the Start button in the resulting popup screen for that station. Note however, that the person who starts up a Station is the only one who can stop it (aside from the system administrators).

7.4 Seed upstream stations with work orders.

There are three ways of doing this:

1. For data-driven processes, an S4P station can often be set up for the directory where the data (or some signal file or delivery record) come into the system. This often involves a special setting in the configuration file to recognize these “foreign” work orders, as they do not follow the normal DO.<job_type>.<job_id>.wo pattern.
2. An alternative for data driven processes is to have a program outside of S4P detect the arrival of data and create a work order for the first upstream station. Such a program may be a run via cron job, triggered by email filter, or forked off by inetd.
3. For ad hoc processing, such as reprocessing, a user interface is used to select the data input for the processing stream and then output the work orders to start the stream processing.

8. Designing Stations and Systems

This section discusses the various options for setting up stations, along with some simple examples.

8.1 S4P Modules For Station Code

S4P includes a number of Perl modules containing many functions that can be used in station code that you write. These are Perl modules, thus we assume here that the scripts making use of them are Perl. Note that the use of the functions and modules described below is completely optional; none have to be used.

S4P Perl Module	Description
S4P.pm	Master module for S4P. For those writing station scripts in Perl, these functions may be particularly useful: • S4P::exec_system() – Executes a system command with error processing. • S4P::logger() – Outputs a message to STDERR in a fixed format. • S4P::perish() – Replacement for die() that calls S4P::logger() before exiting. • S4P::raise_anomaly() – Raises an anomaly by writing a signal file of the given type to the station directory. • S4P::clear_anomaly() – Clears (deletes) a raised anomaly • S4P::restart_job() – Copies all log files and work orders up one directory (so that Station will see the work order and start the job again). • S4P::remove_job() – Removes all files from a failed job directory and deletes the directory. • S4P::stop_station() – Shuts down a station • S4P::snore() – Like sleep() except that it is noisy. It writes a sleep message file. • S4P::repeat_work_order() – Copies the input work order name to its output work order form thereby allowing a suitably configured station to recycle the job.
FileGroup.pm, FileSpec.pm	These modules contain functions for parsing and manipulating Object Data Language (ODL) formatted files such as Product Delivery Records. These modules specifically support work orders written in ODL format. This is a format well-known to those familiar with the EOSDIS Core System (ECS). FileGroup.pm implements a FILE_GROUP object of a PDR where a FILE_GROUP is a group of related files (e.g. Science, Metadata, Production History, and Browse) that make up a data granule. The individual files in the FILE_GROUP are FILE_SPEC objects.
TimeTools.pm	Many functions that parse and manipulate time/date strings.
PCF.pm, PCFEntry.pm	These modules handle ECS-style Process Control Files (PCFs) which has its legacy in algorithms run within ECS. PCFs are used a the work order format in many S4P applications, including S4PM.
ResPool.pm	Module for handling resource pool allocations in a DB_FILE database.
Station.pm	Module for managing Station objects.

8.2 Simple Filters

Perhaps the simplest station is a filter, that is a program that takes a single input file (usually from standard input) and produces a single output file (usually standard output). An example is the Unix **sed** command. The syntax from the command line is usually:

```
command < input > output
```

In an S4P station, however, we anticipate running the same command many times, with different input files, and thus different output files. Therefore, in addition to invoking the command, we must be able to generate the output filename automatically, using say, the time or process_id, or more commonly, a permutation of the input name. One way to do this is to wrap the command in a shell or Perl script.

The following example, lets call it **run_foobar.pl**, runs the command **foobar**, using input files with the pattern `INPUT_xxx.dat` and producing output files with the pattern `OUTPUT_xxx.dat`.

```
#!/usr/bin/perl
$infile = $ARGV[0];
$outfile = $infile;
$outfile =~ s/INPUT/OUTPUT/;
system("foobar < $infile > $outfile");
```

The station configuration file (**station.cfg**) would thus have the following line:

```
%cfg_commands = ('INPUT*.dat'=>'../run_foobar.pl');
```

In this case, **run_foobar.pl** is assumed to be in the station directory. The `'../'` prefix is used because each individual job will be run in a subdirectory created specifically for that job.

This type of script can be used for a number of basic processing setups of the filter type. Note that static arguments could be added to the **foobar** command simply by adding them to the script. For example:

```
system ("foobar -r < $infile > $outfile");
```

8.3 Simple Process Control Files

While simple filters are adequate for many cases, most science algorithms have too many tunable parameters to run with as a simple filter. Often, this problem is solved by process control files. These can vary from simple parameter=value files to large, complex files with specialized syntax, like the ECS Process Control Files. One way to deal with these is to generate a process control template, with all parameters filled in except the input file and output file. Placeholders for the input file and output files (e.g.

INSERT_INPUT_FILE_HERE and INSERT_OUTPUT_FILE_HERE) are then replaced by the station's script on execution:

```
#!/usr/bin/perl
$template_file = '../pcf_template';  # Template is in station directory
open TEMPLATE, $template_file or die "Cannot open $template_file: $!";
open PCF, '>temporary.pcf' or
    die "Cannot open temporary.pcf for writing: $!";
# Generate output filename
$output_file = $ARGV[0];
$output_file =~ s/INPUT/OUTPUT/; # Assumes input file is INPUT_xxx.dat
while (<TEMPLATE>) {
    # Input file is the argument to this script
    s/INSERT_INPUT_FILE_HERE/$ARGV[0]/;
    s/INSERT_OUTPUT_FILE_HERE/$output_file/;
    print PCF;
}
close TEMPLATE;
close PCF;
system("foobar.pl temporary.pcf");
```

8.4 Complex Production Rules

S4P does not have complex production rules built in. However, production rules of arbitrary complexity can be implemented through the use of a triggering station. This is a station whose sole purpose is to create a Process Control File, with all the necessary input and output files filled in. Such a script may access databases or perform complex calculations. Indeed, it may even be an executable binary compiled from C or Fortran. The key criteria for generating such a script are:

(1) It executes using a file as input (i.e., the input work order)
(2) It generates a full-featured process control file, all the information needed for the science algorithm to run
(3) It is self contained, in that with the information in (1) it can generate the process control file on its own

9. S4P Applications

S4P is currently being used for a number of applications at the GES DISC. These are described below.

9.1 S4PM

S4P is the core of the Simple, Scalable, Script-Based, Science Processor for Measurements (S4PM). S4PM comes with already built station code and supports a rich set of production rules. It supports data driven processing as well as on-demand and the Near Archive Data Mining (NADM) system. S4PM has been doing all of the science processing at the GES DISC for four years and has expanded greatly during that time period. S4PM is available on SourceForge at http://sourceforge.net/project/s4pm/.

9.2 S4PA

S4P is also the core of the Simple, Scalable, Script-Based, Science Product Archive (S4PA). S4PA is a data archive management and distribution system for disk-based archives. It currently support about a 400 TB archive at the GES DISC. For more information, see this Website: http://disc.gsfc.nasa.gov/techlab/s4pa/index.shtml.

Appendix A. Acronyms

Acronym	Meaning
AIRS	Atmospheric Infrared Sounder
AVHRR	Advanced Very High-Resolution Radiometer
CPU	Central Processing Unit
DFD	Data Flow Diagram
DISC	Data and Information Services Center
ECS	EOSDIS Core System
EOSDIS	Earth Observing System Data Information System
GES	Goddard Earth Sciences
GFLOPS	Giga FLOPS (Floating Operations Per Second)
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface
HRPT	High Resolution Picture Transmission (AVHRR)
MODIS	Moderate Resolution Imaging Spectroradiometer
ODL	Object Description Language
S4P	Simple, Scalable, Script-Based, Science Processor
SeaWiFS	Sea-viewing Wide Field-of-view Sensor
SIPS	Science Investigator-Led Processing System
TB	Terrabytes

TRMM	Tropical Rainfall Measuring Mission