



**Sonoma Technology, Inc.**  
*Air Quality Research and Innovative Solutions*



# **GETTING STARTED WITH THE BLUESKY FRAMEWORK VERSION 3.0**

**User's Guide  
STI-905028.02-3356-UG**

**By  
Daniel A. Pryden  
Sonoma Technology, Inc.  
1455 N. McDowell Blvd., Suite D  
Petaluma, CA 94954-6503**

**Prepared for  
AirFire Team  
U.S. Department of Agriculture, Forest Service  
400 N. 34th Street, Suite 201  
Seattle, WA 98103**

**With Funding by  
NASA Applied Sciences Program  
Science Mission Directorate  
NASA Headquarters  
Washington, DC 20546**

**May 9, 2008**

## ACKNOWLEDGMENTS

For their assistance in developing this User's Guide, the author gratefully acknowledges the suggestions, technical advice, and feedback offered by the members of the AirFire Team—particularly Narasimhan (Sim) Larkin, Tara Strand, and Robert Solomon—and Dana Sullivan (Sonoma Technology, Inc.).

## TABLE OF CONTENTS

<b><u>Section</u></b>	<b><u>Page</u></b>
ACKNOWLEDGMENTS .....	ii
GETTING STARTED WITH THE BLUESKY FRAMEWORK VERSION 3.0 .....	1
INTRODUCTION.....	1
BLUESKY FRAMEWORK SYSTEM REQUIREMENTS.....	1
INSTALLING THE BLUESKY FRAMEWORK.....	2
USING THE BLUESKY FRAMEWORK .....	2
CONFIGURATIONS AND GRAPHS .....	3
INPUT FILES .....	8
Standard Input Files for Fire Information.....	8
Input Meteorology Files.....	11
CUSTOM MODULES .....	11
Source Code for <code>modules/fccs.py</code> (Example Module).....	13
REFERENCES .....	15

## LIST OF FIGURES

<b><u>Figure</u></b>	<b><u>Page</u></b>
1 Sample excerpt of BlueSky Framework process graph.....	5
2 Example BlueSky Framework configuration file ( <code>config/example.ini</code> ).....	7
3 Choices for the model selections in <code>example.ini</code> , as shown by the <code>bluesky -p</code> command.....	8

## LIST OF TABLES

<b><u>Table</u></b>	<b><u>Page</u></b>
1 Fields in BlueSky Framework standard input and output files.....	9

This page is intentionally blank.

# GETTING STARTED WITH THE BLUESKY FRAMEWORK VERSION 3.0

## INTRODUCTION

The BlueSky Framework is a modeling framework designed to facilitate the operation of predictive models that simulate cumulative smoke impacts, air quality, and emissions from forest, agricultural, and range fires. The BlueSky Framework allows users to combine state-of-the-science emissions, meteorology, and dispersion models to generate results based on the best available models. However, the BlueSky Framework is not itself a model. It is simply a model management system, or framework, that offers the architecture for multiple and varied models to communicate with each other in a modular, user-driven environment.

The BlueSky Framework is an open-source modeling platform that can utilize various model choices at each step. For example, most BlueSky Framework installations have historically used the Emissions Production Model (EPM) created by the USDA Forest Service's (USFS) Fire and Environmental Research Applications (FERA) Team. Version 3.0 of the BlueSky Framework introduces support for new model choices, including the CONSUME 3.0 (USDA Forest Service, 2008a) and FEPS (USDA Forest Service, 2008b) models. BlueSky's main use to date is in modeling fine-scale (less than 2.5 micron) particulate matter. PM<sub>2.5</sub>, as it is known, is a regulated air pollutant under the National Ambient Air Quality Standards.

The BlueSky Framework was created through the close collaboration of land management and air quality regulators with scientific researchers. The BlueSky Framework is governed by the BlueSky Consortium with the USFS AirFire Team taking the lead responsibility for scientific development. Experimental predictions enabled with the BlueSky Framework have been ongoing since 2003. The benefits of the BlueSky Framework are being applied currently by the regional Fire Consortium for the Advanced Modeling of Meteorology and Smoke (FCAMMS) and the National Weather Service.

BlueSky can use a variety of fire information sources, but has also led to development of the SMARTFIRE fire information system (BlueSky Consortium and USDA Forest Service AirFire Team, 2008). SMARTFIRE uses the National Oceanic and Atmospheric Administration's (NOAA) Hazard Mapping System (NOAA Satellite and Information Service, 2008) satellite fire detects with ground reports from systems such as ICS-209 to create a reconciled fire information data feed. SMARTFIRE was developed by the USFS AirFire Team, along with Sonoma Technology, Inc., through a cooperative research agreement funded by the National Aeronautics and Space Administration (NASA).

## BLUESKY FRAMEWORK SYSTEM REQUIREMENTS

The binary build of the BlueSky Framework 3.0 requires the following system specifications:

- Linux 2.2.5 or better; Linux 2.6 or better recommended
- GNU Library (glibc) 2.3.5 or better

- Intel 80386 32-bit or better (64-bit compatible)
- 373MB required for program files; recommend 2GB+ for working space and data
- 1GB memory recommended

Note that using BlueSky 3.0 on a system with SELinux enabled is not currently supported.

## INSTALLING THE BLUESKY FRAMEWORK

For standard installation of the BlueSky Framework 3.0 on a Linux platform, download the binary distribution. The binary distribution is a tarball (`filename.tar.gz`)<sup>1</sup> available at [www.getbluesky.org](http://www.getbluesky.org). After you have downloaded the binary distribution, use the `tar` command to extract the framework—i.e., `tar xzvf bluesky-3.0.0.tar.gz`. (For complex configurations or custom platforms, you may need to build the BlueSky Framework from source, also available at [www.getbluesky.org](http://www.getbluesky.org)).

We recommend that you extract the BlueSky Framework into a standard location, such as `/usr/local/bluesky`. Throughout the remainder of this User’s Guide, we will refer to this directory as `$BS_DIR`. For convenience, we highly recommend that you put `$BS_DIR` in your user `PATH`, which will allow you to start the BlueSky Framework by simply typing “`bluesky`” and can be accomplished by running the following command as the root user:

```
ln -s /usr/local/bluesky/bluesky /usr/bin/bluesky.
```

## USING THE BLUESKY FRAMEWORK

When using the framework, you must supply the name of a configuration that the BlueSky Framework will use as a command-line argument; for example, to start the framework use the following command:

```
bluesky configfilename
```

In practice, most configurations require a date input as follows:

```
bluesky --date yyyyymmdd configfilename
```

where “`--date`” is the switch required by the framework; “`yyyyymmdd`” is the year, month, and day of the run start date; and “`configfilename`” is the name of the configuration file. For example, to run the framework with the configuration file “`example.ini`” for the date “January 12, 2008”, the command to start the framework would be

```
bluesky --date 20080112 example
```

---

<sup>1</sup> A tarball is a compressed archive of files created with the UNIX “`tar`” command and compressed with the “`gzip`” program.

The configuration files are located in “\$BS\_DIR/config” and the “example” configuration file is located here. This configuration fetches fire information remotely from the SMARTFIRE system (BlueSky Consortium and USDA Forest Service AirFire Team, 2008), looks up fuel loading with FCCS (USDA Forest Service, 2008c), calculates consumption with the CONSUME 3.0 (USDA Forest Service, 2008a) model and emissions with the FEPS model (USDA Forest Service, 2008b), specifies a release time profile according to the WRAP method (Air Sciences, 2005), and models the smoke emissions with CALPUFF (TRC Companies, 2008), as well as particle trajectories with HYSPLIT (Air Resources Laboratory, 2008). The outputs from all these models will be placed in a new date-stamped directory under the `output` directory.

Note that, in the above example, we used the “`yyyymmdd`” format for specifying a date. BlueSky actually uses a flexible date format that mostly implements the ISO 8601 “basic” date format. Anywhere BlueSky accepts date information, time information can be provided as well: for example, “`yyyymmddhh`” to specify an hour or “`yyyymmddhhmm`” to specify an hour and minute. Time zones may also be specified, with either a “Z” for UTC, or “L” for an unspecified “local” time zone (time zone will be inferred from context), or an explicit offset from UTC, for example: `200805011058-08:00`. For command-line and configuration file inputs, the time zone is assumed to be UTC if not provided; for fire input files, the time zone is calculated from location or estimated if not provided.

In most places where BlueSky accepts output file names or directories, all the standard `strftime` expansion variables (`%Y`, `%m`, `%d`, etc.) are accepted. For a complete list of expansions on your system, type “`man strftime`”. In most cases where a directory name is expected, you can supply a name with the “@” character, which is replaced by a unique number. For example, in the `default data/defaults.ini`, this technique is used to specify date-specific output folders while ensuring that subsequent runs for the same analysis period do not overwrite each other.

A few additional options exist for debugging or analyzing how the BlueSky Framework works. Many are discussed in Configurations and Graphs below, and further information is available by typing `bluesky -h` on the command line.

## CONFIGURATIONS AND GRAPHS

Central to the BlueSky Framework is the concept of a “process,” which can be any operation that acts on input data to produce output data. The BlueSky Framework works by connecting these processes, each of which can be anything from a simple script to a full-fledged scientific model. The processes can connect to each other in many ways, represented in the BlueSky Framework system as input “plugs” and output “plugs” connecting to form a hierarchy of prerequisites. The BlueSky Framework manages processes by means of a dependency graph.<sup>2</sup>

---

<sup>2</sup> The BlueSky Framework’s dependency graph is a directed acyclic graph (DAG) because the connections between nodes flow in one direction only and because the BlueSky Framework prohibits cyclic dependences or “chicken-and-egg” problems.)

Given a dependency graph, input data, and a target output, the BlueSky Framework can traverse the graph to execute all the dependent tasks in order to produce the desired result.

Generally, the final argument on the BlueSky Framework command line is an execution configuration. The BlueSky Framework loads the corresponding configuration file from `$BS_DIR/config/`, which in turn specifies the dependency graph that the framework will invoke. All the standard configuration files supplied use the same dependency graph, stored in `$BS_DIR/data/default.graph`. This graph contains the dependency information for the standard set of processes and modules included with the BlueSky Framework. Hence, it describes the ways in which these processes will connect and the order in which they will run. The graph defines connections for many output products; however, the configuration may specify only a subset.

If you have the Graphviz visualization suite installed (AT&T Research, 2008), you can display the graph associated with any configuration. The `-G` switch creates an image named `bsf3_graph.jpg` in the current directory, as well as a Graphviz `.dot` file; the `-g` switch also displays the image onscreen. **Figure 1** shows an excerpt of the example graph that is provided with the BlueSky Framework.

If you do not have Graphviz, you can get a similar listing, albeit textually instead of graphically, with the `-l` switch (lowercase letter L, not the number one).

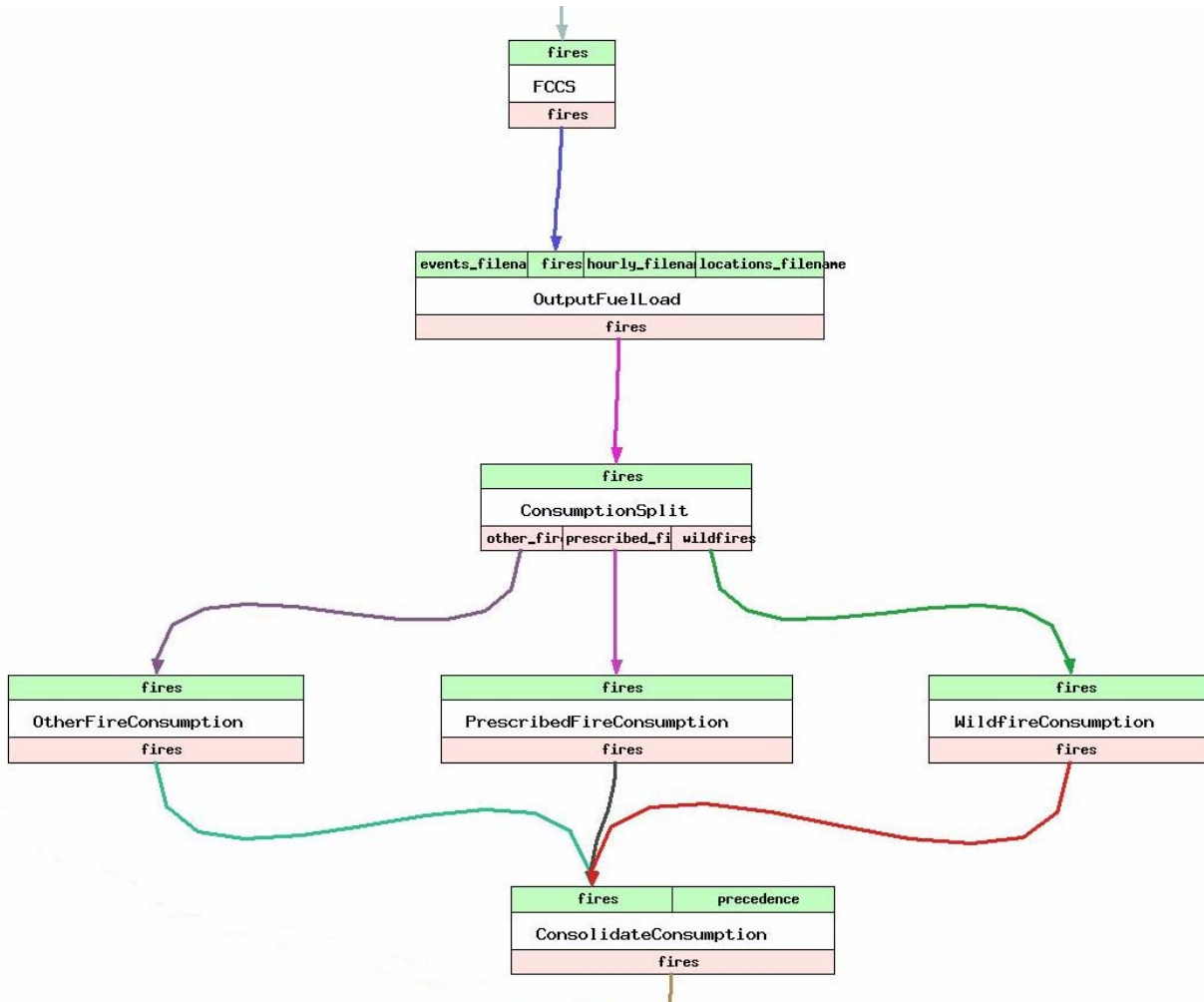


Figure 1. Sample excerpt of BlueSky Framework process graph.

The configuration file selected on the command line should provide a strict hierarchy of tasks, with no alternates (i.e. all steps along the path to any given result should be necessary and sufficient to produce that result). However, the default graph that is provided with the BlueSky Framework actually describes many different possible paths through the system. For convenience, the individual path can be selected at different points by changing a few key variables in the configuration file. Note, however, that this mechanism only selects alternate choices for given places in the graph. To remove a process completely, or replace it with a process that performs a different function, requires changing the graph file. However, this mechanism should be sufficient for most users, so that few will actually have to use a different graph file.

A related concept is the notion of “targets.” Besides specifying a graph with the interconnections between processes, a configuration file should also specify a list of targets that must be executed. Any nodes in the graph that do not contribute to producing the desired targets will not be executed. This constraint facilitates “optional” tasks that will still be included in the graph but will not be executed by default. For example, the BlueSky Framework comes with

some sample configurations, including `example.ini` and `emissionsOnly.ini`, which both share the same graph in `default.graph`. However, the `emissionsOnly.ini` specifies only a subset of the targets that `example.ini` specifies. In general, you probably want to use the predefined “OutputSomething” targets, but any node in the graph can be used as a target.

Though you may specify any exit point for the graph, there is no way to explicitly specify entrance points for the graph. All nodes that are defined in the graph as prerequisites must run before the target node(s) run. However, in practice, many nodes will do little or no work if the input files provided already provide equivalent data. For example, if you provide an input file that already contains consumption information, the consumption model you specify for the graph will not overwrite the existing information. If you want to truly skip prerequisite tasks, you must remove them from the dependency graph so that your targets do not depend on the unwanted nodes.

To use a different configuration of models or tools for several key options along the graph, modify the corresponding values in the example configuration (`$BS_DIR/config/example.ini`, as shown in **Figure 2**), or copy it and invoke it as a different configuration. To see a list of available choices for each configuration option, use the `-process-types (-p)` switch, as in `bluesky --process-types`. An example of its output is shown in **Figure 3**. To see the complete list of all the processes available (even processes for which there are no alternative choices), you can use the `--verbose (-v)` switch, as in `bluesky -p -v`. If you need more flexibility, you can modify the default graph, or else make a new graph of your own.

When the BlueSky Framework starts up, it reads all its configuration inputs and combines them into a single structure; the information from the command-line switches and arguments, the selected configuration file, any graph files that the selected configuration file references, and the special `data/defaults.ini` file are combined into a unified structure internally. Any option that can be specified in one of these places can be specified in any other place. Any default value set up in the `data/defaults.ini` file can be overridden by values in your configuration file. Your configuration file can also override the connections specified in the graph file. Additionally, you can use the `-s` command-line switch to override any of these values.

After creating the internal unified configuration object, the BlueSky Framework immediately saves a copy of that configuration to a new file. This file, called `run.ini`, is stored in the working directory, and saved in the archive tarball that is created when the run is complete. This file contains all the information from all the configuration files that the BlueSky Framework used when initializing for the current run, with a comment next to each item listing the file where the value was defined. Additionally, this is a valid BlueSky Framework initialization file, so you can use it to invoke the BlueSky Framework again, with the `-i` switch.

```

#####
#
# BlueSky Framework - Controls the estimation of emissions, incorporation of
# meteorology, and the use of dispersion models to
# forecast smoke impacts from fires.
# Copyright (C) 2003-2006 USDA Forest Service - Pacific Northwest Wildland
# Fire Sciences Laboratory
# BlueSky Framework - Version 3.0
# Copyright (C) 2007 USDA Forest Service - Pacific Northwest Wildland Fire
# Sciences Laboratory and Sonoma Technology, Inc.
# All rights reserved.
#
# See LICENSE.TXT for the Software License Agreement governing the use of the
# BlueSky Framework - Version 3.0.
#
# Contributors to the BlueSky Framework are identified in ACKNOWLEDGEMENTS.TXT
#
#####

#
# Example BlueSky configuration file. This is the file that end-users should
# modify directly. Advanced users can customize the behavior by modifying
# the dependency graph in default_graph.ini or creating a new graph.
#

[META]
includes=${BS_DIR}/data/default.graph
#
# What are our default targets? (i.e. what should this config do by default?)
#
targets=OutputEmissions OutputSmokeFiles OUTPUT_$DISPERSION OUTPUT_$TRAJECTORY

[DEFAULT]
#
# Analysis time range (override with -d command-line switch)
#
DATE = 2008050100Z
EMISSIONS_OFFSET = 0
DISPERSION_OFFSET = 0
HOURS_TO_RUN = 48
HOURS_TO_RUN_TRAJECTORY = 12

#
# What models are we going to run to get those outputs?
#
INPUTS=StandardFiles,SMARTFIRE
FUEL_LOAD=FCCS
WILDFIRE_CONSUMPTION=CONSUME
PRESCRIBED_CONSUMPTION=CONSUME
OTHER_CONSUMPTION=CONSUME
TIME_PROFILE=WRAPtimeProfile
EMISSIONS=FEPSEmissions
PLUME_RISE=WRAPplumeRise
DISPERSION=CALPUFF
TRAJECTORY=HYSPLIT

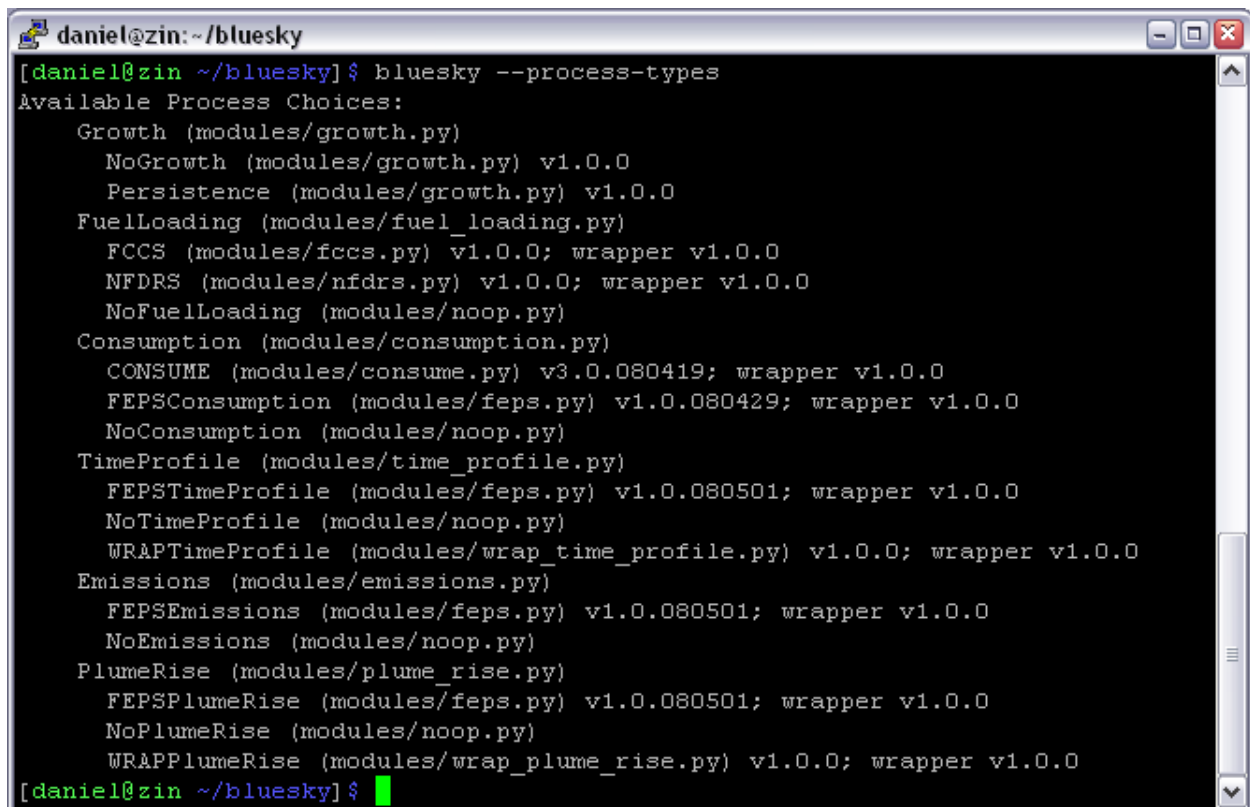
# Growth models
WILDFIRE_GROWTH=Persistence
PRESCRIBED_GROWTH=NoGrowth
OTHER_GROWTH=NoGrowth

# Consumption canopy fraction
WILDFIRE_CANOPY_FRACTION = auto
PRESCRIBED_CANOPY_FRACTION = 0.0
OTHER_CANOPY_FRACTION = auto

#
# Other run-specific options
#
STOP_IF_NO_BURNS = true

```

Figure 2. Example BlueSky Framework configuration file (config/example.ini).



```
daniel@zin:~/bluesky
[daniel@zin ~/bluesky]$ bluesky --process-types
Available Process Choices:
  Growth (modules/growth.py)
    NoGrowth (modules/growth.py) v1.0.0
    Persistence (modules/growth.py) v1.0.0
  FuelLoading (modules/fuel_loading.py)
    FCCS (modules/fccs.py) v1.0.0; wrapper v1.0.0
    NFDRS (modules/nfdrs.py) v1.0.0; wrapper v1.0.0
    NoFuelLoading (modules/noop.py)
  Consumption (modules/consumption.py)
    CONSUME (modules/consume.py) v3.0.080419; wrapper v1.0.0
    FEPSConsumption (modules/feps.py) v1.0.080429; wrapper v1.0.0
    NoConsumption (modules/noop.py)
  TimeProfile (modules/time_profile.py)
    FEPSTimeProfile (modules/feps.py) v1.0.080501; wrapper v1.0.0
    NoTimeProfile (modules/noop.py)
    WRAPTimeProfile (modules/wrap_time_profile.py) v1.0.0; wrapper v1.0.0
  Emissions (modules/emissions.py)
    FEPSEmissions (modules/feps.py) v1.0.080501; wrapper v1.0.0
    NoEmissions (modules/noop.py)
  PlumeRise (modules/plume_rise.py)
    FEPSPlumeRise (modules/feps.py) v1.0.080501; wrapper v1.0.0
    NoPlumeRise (modules/noop.py)
    WRAPPlumeRise (modules/wrap_plume_rise.py) v1.0.0; wrapper v1.0.0
[daniel@zin ~/bluesky]$
```

Figure 3. Choices for the model selections in `example.ini`, as shown by the `bluesky -p` command.

Aside from the normal command-line argument invocation method (e.g., `bluesky example`), you can invoke any configuration file by specifying it with the `-i` switch, and you can use the `-i` switch to specify multiple configuration files, which are each loaded in turn. However, if the configuration does not specify a default set of targets in the `[META]` section, you need to use the `-r` switch to select a target to execute. You can also use the `-r` switch multiple times to specify multiple desired outputs; these outputs will all be produced.

Additionally, arbitrary configuration values can be adjusted on the command-line by using the `-s` switch. For example, to change the CALPUFF binary used by a single run of The BlueSky Framework, you could specify `-sCALPUFF.CALPUFF_BINARY=/usr/local/calpuff/bin/calpuff`. Note that the configuration file is generally easier to use than this method.

## INPUT FILES

### Standard Input Files for Fire Information

In addition to on-demand data downloads from the SMARTFIRE system, the BlueSky Framework supports input of fire information through a standard CSV format. By default, input files should be stored in the `$BS_DIR/input/fires/` directory.

The BlueSky Framework standard input file format is intended to be compatible with the output files produced by the Framework. The input data format is actually made up of three files, two of which are optional for input. Each is a comma-delimited file with any number of columns and rows. The first line of the file must list the names of the columns. Columns in data records can be in any order as long as the data fields match up with the columns. The BlueSky Framework looks for a standard set of columns for values, but additional fields may be present. They are stored by the BlueSky Framework in a structure called “metadata” and are preserved to be written in the output file. **Table 1** lists the fields that the BlueSky Framework recognizes.

Table 1. Fields in BlueSky Framework standard input and output files

File	Column	Type	Description
fire_events.csv	id	Required Input	Unique identifier for this fire event
	event_name	Required Input	Event Name
	total_area	Derived	Sum of locations' area for the entire period (acres)
	total_heat	Derived	Sum of locations' heat for the entire period (BTU)
	total_pm25	Derived	Sum of locations' PM <sub>2.5</sub> for the entire period (tons)
	total_pm10	Derived	Sum of locations' PM <sub>10</sub> for the entire period (tons)
	total_pm	Derived	Sum of locations' PM for the entire period (tons)
	total_co	Derived	Sum of locations' CO for the entire period (tons)
	total_co2	Derived	Sum of locations' CO <sub>2</sub> for the entire period (tons)
	total_ch4	Derived	Sum of locations' CH <sub>4</sub> for the entire period (tons)
	total_nmhc	Derived	Sum of locations' NMHC for the entire period (tons)
	total_nox	Derived	Sum of locations' NO <sub>x</sub> for the entire period (tons)
	total_nh3	Derived	Sum of locations' NH <sub>3</sub> for the entire period (tons)
	total_so2	Derived	Sum of locations' SO <sub>2</sub> for the entire period (tons)
<all others>	Metadata	Other event metadata (pass-through)	
fire_locations.csv	id	Required Input	Unique identifier for this fire location (any format)
	event_id	Optional	Event ID to look up in fire_events.csv
	latitude	Required Input	Location of this fire (latitude in decimal degrees)
	longitude	Required Input	Location of this fire (longitude in decimal degrees)
	elevation	Optional	Elevation at fire (meters)
	slope	Optional	Slope at fire location
	state	Optional/Derived	Location information
	county	Optional	Location information
	country	Optional/Derived	Location information
	date_time	Required Input	Time fire occurred (ignition time or local midnight)
	duration	Optional/Derived	Ignition duration
	snow_month	Optional	Month of last snow (rarely provided)
	rain_days	Optional	Days since last rain (rarely provided)
	wind	Optional/Derived	Wind (mph) at fire (rarely provided)
	type	Optional/Derived	Type of this fire: WF=wildfire, WFU=wildland fire use, RX=prescribed or Unknown
	area	Required Input	Area of this fire in acres
	fuel_1hr	Optional/Derived	Total loading of 1-hr fuels (tons)
	fuel_10hr	Optional/Derived	Total loading of 10-hr fuels (tons)
	fuel_100hr	Optional/Derived	Total loading of 100-hr fuels (tons)
	fuel_1khr	Optional/Derived	Total loading of 1,000-hr fuels (tons)
fuel_10khr	Optional/Derived	Total loading of 10,000-hr fuels (tons)	

	fuel_gt10hr	Optional/Derived	Total loading of > 10,000-hr fuels (tons)
	shrub	Optional/Derived	Total loading of shrub fuels (tons)
	grass	Optional/Derived	Total loading of grassy fuels (tons)
	rot	Optional/Derived	Total loading of rotted fuels (tons)
	duff	Optional/Derived	Depth of fuel loading in the duff layer (inches)
	litter	Optional/Derived	Total loading of litter fuels (tons)
	fuel_moisture_10hr	Optional/Derived	Moisture (%) of 10-hr fuel
	fuel_moisture_1hr	Optional/Derived	Moisture (%) of 1000-hr fuel
	consumption_flaming	Optional/Derived	Total consumption in the flaming phase (tons)
	consumption_smoldering	Optional/Derived	Total consumption in the smoldering phase (tons)
	consumption_residual	Optional/Derived	Total consumption in the residual phase (tons)
	consumption_duff	Optional/Derived	Total consumption of the duff layer (tons)
	heat	Optional/Derived	Total heat, or sum of hourly heat released (BTU)
	pm25	Optional/Derived	Total PM <sub>2.5</sub> , or sum of hourly PM <sub>2.5</sub> (tons)
	pm10	Optional/Derived	Total PM <sub>10</sub> , or sum of hourly PM <sub>10</sub> (tons)
	pm	Optional/Derived	Total PM, or sum of hourly PM (tons)
	co	Optional/Derived	Total CO, or sum of hourly CO (tons)
	co2	Optional/Derived	Total CO <sub>2</sub> , or sum of hourly CO <sub>2</sub> (tons)
	ch4	Optional/Derived	Total CH <sub>4</sub> , or sum of hourly CH <sub>4</sub> (tons)
	nmhc	Optional/Derived	Total NMHC, or sum of hourly NMHC (tons)
	nox	Optional/Derived	Total NO <sub>x</sub> , or sum of hourly NO <sub>x</sub> (tons)
	nh3	Optional/Derived	Total NH <sub>3</sub> , or sum of hourly NH <sub>3</sub> (tons)
	so2	Optional/Derived	Total SO <sub>2</sub> , or sum of hourly SO <sub>2</sub> (tons)
	<all others>	Metadata	Location metadata (pass-through)
<b>fire_hourly.csv</b>	fire_id	Required	Location ID to look up in fire_locations.csv
	hour	Required	Hour of data (counted from start date_time)
	area_fract	Required	Area fraction for the given hour (total sums to 1)
	flame_profile	Required	Flaming fraction for the hour (total sums to 1)
	smolder_profile	Required	Smoldering fraction for the hour (total sums to 1)
	residual_profile	Required	Residual fraction for the hour (total sums to 1)
	heat_emitted	Derived	Total emission this hour of heat (BTU)
	pm25_emitted	Derived	Total emission this hour of PM <sub>2.5</sub> (tons)
	pm10_emitted	Derived	Total emission this hour of PM <sub>10</sub> (tons)
	pm_emitted	Derived	Total emission this hour of PM (tons)
	co_emitted	Derived	Total emission this hour of CO (tons)
	co2_emitted	Derived	Total emission this hour of CO <sub>2</sub> (tons)
	ch4_emitted	Derived	Total emission this hour of CH <sub>4</sub> (tons)
	nmhc_emitted	Derived	Total emission this hour of NMHC (tons)
	nox_emitted	Derived	Total emission this hour of NO <sub>x</sub> (tons)
	nh3_emitted	Derived	Total emission this hour of NH <sub>3</sub> (tons)
	so2_emitted	Derived	Total emission this hour of SO <sub>2</sub> (tons)
	pm25_flame	Derived	Flaming emission this hour of PM <sub>2.5</sub> (tons)
	pm10_flame	Derived	Flaming emission this hour of PM <sub>10</sub> (tons)
	pm_flame	Derived	Flaming emission this hour of PM (tons)
	co_flame	Derived	Flaming emission this hour of CO (tons)
	co2_flame	Derived	Flaming emission this hour of CO <sub>2</sub> (tons)
	ch4_flame	Derived	Flaming emission this hour of CH <sub>4</sub> (tons)
	nmhc_flame	Derived	Flaming emission this hour of NMHC (tons)
	nox_flame	Derived	Flaming emission this hour of NO <sub>x</sub> (tons)
	nh3_flame	Derived	Flaming emission this hour of NH <sub>3</sub> (tons)

so2_flame	Derived	Flaming emission this hour of SO <sub>2</sub> (tons)
pm25_smold	Derived	Smoldering emission this hour of PM <sub>2.5</sub> (tons)
pm10_smold	Derived	Smoldering emission this hour of PM <sub>10</sub> (tons)
pm_smold	Derived	Smoldering emission this hour of PM (tons)
co_smold	Derived	Smoldering emission this hour of CO (tons)
co2_smold	Derived	Smoldering emission this hour of CO <sub>2</sub> (tons)
ch4_smold	Derived	Smoldering emission this hour of CH <sub>4</sub> (tons)
nmhc_smold	Derived	Smoldering emission this hour of NMHC (tons)
nox_smold	Derived	Smoldering emission this hour of NO <sub>x</sub> (tons)
nh3_smold	Derived	Smoldering emission this hour of NH <sub>3</sub> (tons)
so2_smold	Derived	Smoldering emission this hour of SO <sub>2</sub> (tons)
pm25_resid	Derived	Residual emission this hour of PM <sub>2.5</sub> (tons)
pm10_resid	Derived	Residual emission this hour of PM <sub>10</sub> (tons)
pm_resid	Derived	Residual emission this hour of PM (tons)
co_resid	Derived	Residual emission this hour of CO (tons)
co2_resid	Derived	Residual emission this hour of CO <sub>2</sub> (tons)
ch4_resid	Derived	Residual emission this hour of CH <sub>4</sub> (tons)
nmhc_resid	Derived	Residual emission this hour of NMHC (tons)
nox_resid	Derived	Residual emission this hour of NO <sub>x</sub> (tons)
nh3_resid	Derived	Residual emission this hour of NH <sub>3</sub> (tons)
so2_resid	Derived	Residual emission this hour of SO <sub>2</sub> (tons)

## Input Meteorology Files

For certain operations, particularly dispersion and trajectory calculations, gridded meteorological data are necessary. The directory and filename pattern are stored in the `data/defaults.ini` file. By default, the `data/defaults.ini` file is set up so that the framework expects MM5-format meteorological data to be stored in the `$BS_DIR/input/met/` directory, in files named `MMOUT_DOMAIN1.yyyymmdd.hhz.n`, where `yyymmdd` is the year, month, and day of the data, `hh` is the model start hour, and `n` is a sequence number of the MM5 output files. The directory and file name pattern can be changed in the `data/defaults.ini` file or overridden in your configuration file or on the command line.

## CUSTOM MODULES

Version 3.0 of the BlueSky Framework is implemented in version 2.5 of the Python programming language. For information about the Python language, see <http://www.python.org> or refer to a Python reference work. The remainder of this section assumes familiarity with Python and object-oriented programming concepts, such as objects, classes, and subclasses.

The processes used in the dependency graph (see Configurations and Graphs, above) are actually instances of Python objects, which are defined in module files. There are two types of modules: a set of “base” modules is shipped with the BlueSky Framework and provides processes that perform system tasks or internal configuration; all other modules are stored as Python files in the `$BS_DIR/modules` directory. The Python modules are available in two forms: the `.py` files contain the Python source code to the modules, and the `.pyc` files contain

compiled Python bytecode. The BlueSky Framework will automatically recompile any modules that are changed or added into the system, re-creating the `.pyc` files as needed.

To prepare a custom module, you can use an existing module as a template. Most new processes should be built as subclasses of existing base processes, which set up the environment for a given type of process. For example, to add a new fuel loading option, use the `FCCS` class defined in `$BS_DIR/modules/fccs.py` (see Source Code to `modules/fccs.py`, below) as a template, and make your class a subclass of the generic `FuelLoading` class defined in `$BS_DIR/modules/fuel_loading.py`.

All processes must declare an `init()` method that calls `self.declare_input()` and `self.declare_output()` to set up its input and output connection “plugs”. This method allows the process to be used in a graph.<sup>3</sup> The process must also declare a `run()` method that takes the current execution context as an argument. The `run` method can retrieve its input objects using `self.get_input()` and can retrieve configuration information from its configuration file section using `self.config()`. It can manipulate these objects, but it must call `self.set_output()` when complete to set any declared outputs.

Log messages can be output through the `self.log` object, which is a `logging.Logger` object from the Python standard library. External processes can be launched by using the `context.execute()` method (the `context` object is passed as an argument to the `run()` method). Always use the `context.full_path()` method to resolve relative paths, since these are relative to the current execution context in the working directory.

Many standard library modules are available inside the BlueSky Framework Python execution environment. However, because the BlueSky Framework is a frozen Python distribution, some modules cannot be imported, and others may behave in unexpected ways. It is usually best to use the Python module code as a “wrapper” around external C, C++, Fortran, or other code, so that the framework can gracefully recover if the external process fails because of an error condition.

If an error occurs inside any module, the BlueSky Framework will terminate and clean up any files in its working directory. Debugging information is stored in the `run.log` file, which is archived in a tarball stored in the output directory. If the error was a Python exception, a stack trace will be printed, which will provide more information about the error and its location in the source code.

---

<sup>3</sup> These connections may also be declared in a superclass. For example, any class derived from the generic `FuelLoading` class has the same set of connections as any other `FuelLoading` class, so that they are interchangeable.

## Source Code for `modules/fccs.py` (Example Module)

```
*****
#
# BlueSky Framework - Controls the estimation of emissions, incorporation of
#                       meteorology, and the use of dispersion models to
#                       forecast smoke impacts from fires.
# Copyright (C) 2003-2006  USDA Forest Service - Pacific Northwest Wildland
#                       Fire Sciences Laboratory
# BlueSky Framework - Version 3.0
# Copyright (C) 2007  USDA Forest Service - Pacific Northwest Wildland Fire
#                       Sciences Laboratory and Sonoma Technology, Inc.
#                       All rights reserved.
#
# See LICENSE.TXT for the Software License Agreement governing the use of the
# BlueSky Framework - Version 3.0.
#
# Contributors to the BlueSky Framework are identified in ACKNOWLEDGEMENTS.TXT
#
*****

import csv
from Scientific.IO.NetCDF import NetCDFFile
import Numeric

from fuel_loading import FuelLoading
from kernel.types import construct_type
from kernel import projection

class FCCS(FuelLoading):
    """ FCCS Fuel Loading Module """

    def run(self, context):

        #
        # Get fires from input plug
        #
        fireInfo = self.get_input("fires")

        #
        # Get NetCDF and CSV 'library' filenames (rarely changed)
        #
        FUEL_LOAD_DATA = self.config("FUEL_LOAD_DATA")
        FUEL_LOOKUP_CSV = self.config("FUEL_LOOKUP_CSV")

        #
        # Open the NetCDF file and extract the fuel loading
        #
        ncfile = NetCDFFile(FUEL_LOAD_DATA, 'r')
        projectXY = projection.getIoapiProjection(ncfile)
        fccsData = ncfile.variables["fccfuel"]

        #
        # Read in the CSV Fuel Lookup data into a dictionary for easy lookup
        #
        fccsLoading = {}
        for row in csv.DictReader(file(FUEL_LOOKUP_CSV, 'r')):
            fccsLoading[int(row["mapID"])] = row

        timestep, lay, nrows, ncols = Numeric.shape(fccsData)
```

```

nFires = len(fireInfo.locations())
nSuccess = 0
for fireLoc in fireInfo.locations():

    if fireLoc["fuels"] is not None:
        self.log.info( "Skipping %s, fuels already set" % fireLoc)
        continue

    fuelInfo = construct_type("FuelsData")
    fireLoc["fuels"] = fuelInfo

    fuelInfo["fccs_number"] = 0
    fuelInfo["fuel_1hr"] = 0.0
    fuelInfo["fuel_10hr"] = 0.0
    fuelInfo["fuel_100hr"] = 0.0
    fuelInfo["fuel_1khr"] = 0.0
    fuelInfo["fuel_10khr"] = 0.0
    fuelInfo["fuel_gt10khr"] = 0.0
    fuelInfo["shrub"] = 0.0
    fuelInfo["grass"] = 0.0
    fuelInfo["rot"] = 0.0
    fuelInfo["duff"] = 0.0
    fuelInfo["litter"] = 0.0

    if fireLoc["latitude"] is None or fireLoc["longitude"] is None:
        self.log.debug("Invalid location for %s, "
                       "specifying zero fuelload", fireLoc)
        continue

    (xgrid, ygrid) = projectXY(fireLoc["latitude"],
                               fireLoc["longitude"])

    if (ygrid < 0) or (ygrid >= nrows):
        self.log.debug("Error: %s y-dim outside domain, "
                       "specifying zero fuelload ", fireLoc)
        continue
    if (xgrid < 0) or (xgrid >= ncols):
        self.log.debug("Error: %s x-dim outside domain, "
                       "specifying zero fuelload ", fireLoc)
        continue

    fuelModel = fccsData[0,0,ygrid,xgrid].toscalar()

    if fuelModel < 0 or fuelModel > 289:
        self.log.debug("No data for %s, fuelModel = %d",
                       fireLoc, fuelModel)
        continue

    self.log.debug("%s: FCCS Fuel Loading: %s",
                   fireLoc, fccsLoading[fuelModel])
    nSuccess += 1

    fuelInfo["fccs_number"] = fuelModel

    fuelInfo["fuel_1hr"] = float(fccsLoading[fuelModel]["1HR"])
    fuelInfo["fuel_10hr"] = float(fccsLoading[fuelModel]["10HR"])
    fuelInfo["fuel_100hr"] = float(fccsLoading[fuelModel]["100HR"])
    fuelInfo["fuel_1khr"] = float(fccsLoading[fuelModel]["1kHR"])
    fuelInfo["fuel_10khr"] = float(fccsLoading[fuelModel]["10kHR"])
    fuelInfo["fuel_gt10khr"] = float(fccsLoading[fuelModel]["10k+HR"])
    fuelInfo["shrub"] = float(fccsLoading[fuelModel]["SHRUB"])
    fuelInfo["grass"] = float(fccsLoading[fuelModel]["GRASS"])
    fuelInfo["rot"] = 1

```

```

fuelInfo["duff"] = float(fccsLoading[fuelModel]["DUFF"])
fuelInfo["litter"] = 14
fuelInfo["metadata"]["VEG"] = fccsLoading[fuelModel]["VEG"]

# Close the NetCDF file
ncfile.close()

self.log.info("Successfully added fuel loadings "
             "for %d of %d input fires", nSuccess, nFires)

#
# Set output plug to updated fireInfo
#
self.set_output("fires", fireInfo)

```

## REFERENCES

- Air Resources Laboratory (2008) HYSPLIT model. Web page of the National Oceanic and Atmospheric Association, Air Resources Laboratory, Silver Spring, MD. Available on the Internet at <<http://www.arl.noaa.gov/ready/hysplit4.html>>.
- Air Sciences, Inc., (2005) 2002 Fire emission inventory for the WRAP Region – Phase II. Report prepared for the Western Governors Association/Western Regional Air Partnership, Denver, CO/Ronan, MT, by Air Sciences, Inc., Denver, CO/Portland, OR, Project No. 178-6, July. Available on the Internet at <[http://www.wrapair.org/forums/fejf/documents/WRAP\\_2002\\_PhII\\_EI\\_Report\\_20050722.pdf](http://www.wrapair.org/forums/fejf/documents/WRAP_2002_PhII_EI_Report_20050722.pdf)>.
- AT&T Research (2008) Graphviz - graph visualization software. Available on the Internet at <<http://www.graphviz.org/>>.
- BlueSky Consortium and USDA Forest Service AirFire Team (2008) SMARTFIRE, Satellite Mapping Automatic Reanalysis Tool for Fire Incident Reconciliation. Available on the Internet at <<http://getbluesky.org/smartfire/>>.
- NOAA Satellite and Information Service (2008) Hazard Mapping System Fire and Smoke Product. Available on the Internet at <<http://www.ssd.noaa.gov/PS/FIRE/hms.html>>.
- TRC Companies, Inc., (2008) The CALPUFF modeling system. Web page of the Atmospheric Studies Group at TRC Companies, Inc., Lowell, MA. Available on the Internet at <<http://www.src.com/calpuff/calpuff1.htm>>.
- USDA Forest Service (2008a) CONSUME. Web page of the U.S. Forest Service, Fire and Environmental Research Applications Team, Seattle, WA. Available on the Internet at <<http://www.fs.fed.us/pnw/fera/research/smoke/consume/index.shtml>>.
- USDA Forest Service (2008b) Fire Emission Production Simulator (FEPS). Web page of the U.S. Forest Service, Fire and Environmental Research Applications Team, Seattle, WA. Available on the Internet at <<http://www.fs.fed.us/pnw/fera/feps/>>.

USDA Forest Service (2008c) The Fuel Characteristic Classification System (FCCS). Web page of the U.S. Forest Service, Fire and Environmental Research Applications Team, Seattle, WA. Available on the Internet at <<http://www.fs.fed.us/pnw/fera/fccs/>>.