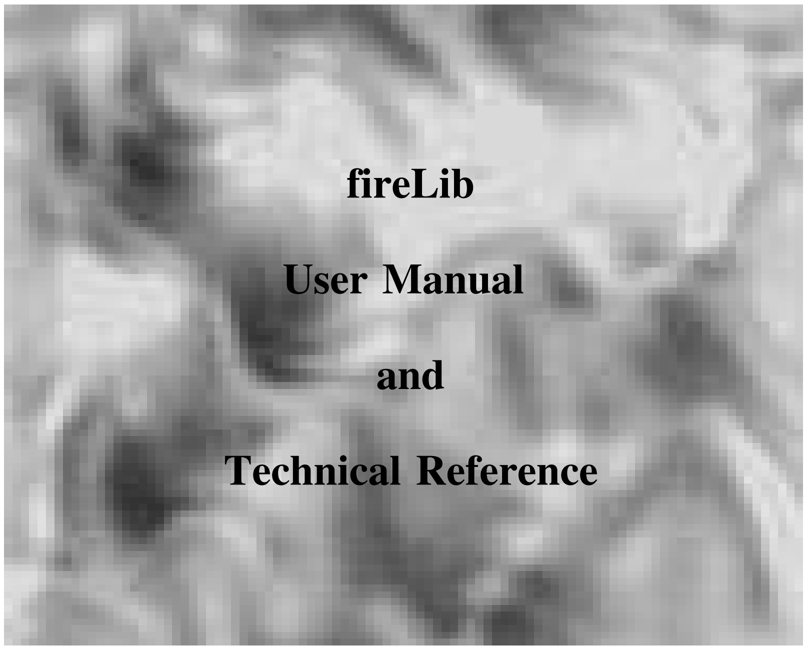

fireLib User Manual and Technical Reference

October 1996

Collin D. Bevins
Systems for Environmental Management

fireLib User Manual and Technical Reference

Release 1, October 1996

This research was supported in part by funds provided by the Intermountain Research Station, Forest Service, U.S. Department of Agriculture. Additional support was provided by Systems for Environmental Management, a Montana nonprofit research organization.

0. Introduction

fireLib is a C function library for predicting the spread rate and intensity of free-burning wildfires. It is derived directly from the BEHAVE (Andrews 1986) fire behavior algorithms for predicting fire spread in two dimensions, but is optimized for highly iterative applications such as cell- or wave-based fire growth simulation.

The BEHAVE System was developed in the early 1980's to calculate a few (less than 50) fire behavior projections at a time and display the results in a small table. In the past decade greater attention has been given to spatially explicit fire growth modeling where fire is simulated within a large array of terrain, fuel, and weather conditions that vary spatially and temporally. Such growth models make highly repetitive computations of fire spread from each source point to multiple destination points; a process that is repeated for each source point in the map and for each time step in the simulation. This quickly results in millions of iterations and provides an opportunity to realize significant benefits from a library of optimally encapsulated fire behavior functions.

The fireLib library was developed to give fire growth modellers a simple, common, and optimized application programming interface (API) to use in their applications. It is written entirely in ANSI standard C and also compiles under a wide range of Kernighan & Ritchie (pre-ANSI standard) C and C++ compilers on a variety of personal computers and work stations.

While fireLib contains 13 functions, as few as 4 functions are required to create a simple yet efficient and functional fire growth simulator. The example simulator may serve as a foundation for more realistic applications.

This paper is both a user manual and a technical reference to fireLib. We assume the reader is familiar with the basic concepts of fuel models and fire behavior modeling and make no attempt to teach these concepts. Readers are furthermore cautioned to understand the assumptions and limitations (summarized by Andrews 1986) of the Rothermel (1972) fire model, the BEHAVE System, and all models derived from them, including the fireLib API.

1. The Fire Behavior Calculation Pipeline

Calculation of fire spread and intensity may be represented as a pipeline through which sets of input parameters are introduced at four stages to calculate succeeding fire behavior variables:

Stage	Stage Inputs	Stage Outputs
1: Fuel	Fuel bed and fuel particle characteristics	Characteristic fuel area, load, etc; fuel bed bulk density; fire residence time.
2: Moisture	Fuel moisture	No-wind, no-slope spread rate; reaction intensity; heat per unit area.; live fuel extinction moisture.
3: Wind	Wind speed & direction, slope & aspect	Maximum spread rate; direction of maximum spread.
4: Direction	Spread azimuth	Spread rate; fireline intensity; flame length; scorch height (all in the direction of interest)

While fuel bed characteristics generally vary throughout the landscape, they are usually considered invariant within the time frame of a fire behavior growth simulation. A significant improvement in fire modelling performance is therefore realized by calculating and storing variables that are fuel bed dependent intermediate for each fuel model. In the first stage of the pipeline variables such as characteristic surface area, loading, and residence time are derived. The fireLib `Fire_FuelCombustion`() function performs the first stage computations.

Fuel moisture, wind speed, and wind direction are more temporal. Because the Rothermel (1972) fire model uses fuel moisture to determine the heat sink term in deriving fire reaction intensity and the no-wind no-slope spread rate, fuel moisture is introduced in the second stage of the pipeline. The fireLib `Fire_SpreadNoWindNoSlope`() function performs the second stage computations.

Rothermel (1972) modelled the effect of slope on fire behavior using the same mechanism as for wind; in fact he calls the combined effect of slope and wind the "effective windspeed". Thus, although slope and aspect are invariant over time, they are introduced into the pipeline at stage 3 along with wind speed and wind direction. In the third stage the maximum spread rate and direction of maximum spread are calculated by the fireLib `Fire_SpreadWindSlopeMax`() function.

In Stage 4, fire behavior in any compass direction is determined using the elliptical growth model developed by Anderson (1983) and employed by BEHAVE (Andrews 1986). The four vector-dependent fire behavior outputs of spread rate, Byram's fireline intensity, flame length, and scorch height are derived by the fireLib `Fire_SpreadAtAzimuth`() function.

Significant improvement in fire model performance is realized by partitioning the fire behavior computations into these four stages and proceeding only through those stages necessary to arrive at new behavior estimates.

After optimizing the fire behavior algorithms from BEHAVE, the following proportion of time is spent in a single iteration through each stage of the pipeline:

Stage	Computations	Time (%)
1	Fuel bed and combustion intermediates	82
2	No-wind, no-slope spread rate	2
3	Maximum spread rate & direction of maximum spread	14
4	Spread rate, intensity, flame length, & scorch height in any direction	2

2. Fuel Catalog, Fuel Model, and Fuel Particle Objects

The fireLib API employs the concepts of a fuel catalog, a fuel model, and a fuel particle. A fireLib application may contain one or more fuel catalogs. A fuel catalog is a collection of zero or more fire behavior fuel models, and each fuel model contains zero or more fuel particles.

2.1 Fuel Catalog Objects and Methods

A *fuel catalog* is merely a collection of fuel models. In addition to fuel data, it serves as a container for state information on function status, error messages, and use of the optional flame length table. Each fuel catalog is identified by a *handle* (or pointer) returned by one of the fuel catalog creation functions. All other functions require this handle as an input argument.

The `Fire_FuelCatalogCreate()` function creates a new fuel catalog. The catalog name and maximum number of fuel models it may contain are specified as function arguments. The fuel models are subsequently defined for the catalog by the `Fire_FuelModelCreate()` function (see below).

The `Fire_FuelCatalogCreateStandard()` function also creates a new fuel catalog with the specified name and fuel model capacity. It also automatically pre-defines 14 fuel models; a no-fuel model and the 13 standard fire behavior models.

The `Fire_FuelCatalogDestroy()` function removes the specified fuel catalog and all its fuel models from program memory.

2.2 Fuel Model Objects and Methods

A *fuel model* describes the fuel particles and fuel bed arrangement for a stylized or representative fuel condition. The fuel model object also serves as a container for all the current fire environment, fuel bed, and fire behavior state information relating to the fuel model.

Each fuel model within a catalog is identified by a unique integer number. This number may range from zero to the maximum number of models allowed in the catalog. All fireLib fire behavior functions require the fuel catalog handle and the fuel model number as input arguments.

The `Fire_FuelModelCreate()` function creates a new fuel model. The catalog handle, model number, name, description, maximum number of fuel particles allowed, fuel bed depth, and dead fuel extinction moisture content are specified as function arguments.

This function will also re-define any fuel model with the same model number in the catalog. Fuel particles are subsequently added to the fuel model by the `Fire_FuelParticleAdd`() function (see below).

The `Fire_FuelModelDestroy()` function removes a fuel model and its fuel particles from its catalog.

It is important to note that because each fuel catalog is a separate object, there is not necessarily any correspondence between fuel models with the same numbers in different catalogs. For example, fuel model 6 in one fuel catalog may be entirely different from fuel model 6 in another catalog, depending on how the fuel models were created.

2.3 Fuel Particle Objects and Methods

A *fuel particle* is a specific type of fuel component found in a fuel model. Examples of fuel particles include needle, grass, or foliage litter; standing grass; dead and down twigs and branches of various diameters; and live stems and foliage.

The `Fire_FuelParticleAdd`() function adds new fuel particles to a fuel model within a catalog. Function arguments include the particle life code (dead, live herbaceous, or live woody stem), surface-area-to-volume ratio, loading, particle density, low heat of combustion, and total and effective silica content. Each fuel model may contain from zero to the maximum number of fuel particles specified when the fuel model was created.

Fuel particles are assigned an index number (starting with 0) as they are added to the fuel model. Programmers use the fuel particle property macros (section 5.2) to access or update particle characteristics. There is no function for removing fuel particles from a fuel model.

2.4 Flame Length Table

The flame length table is an optional object belonging to the fuel catalog.

The `Fire_SpreadAtAzimuth`() and `Fire_FlameScorch`() functions calculate flame length using a power function of the fireline intensity. The power function can significantly impact program performance, and it is generally more efficient to look up the flame length from a pre-calculated table given the fireline intensity. Since flame length is strictly an output variable (it is never used to derive any other variable), it is usually not necessary to calculate it to full floating point precision anyway.

The `Fire_FlameLengthTable`() builds a flame length look up table that is subsequently used by `Fire_SpreadAtAzimuth`() and `Fire_FlameScorch`() instead of direct computation. The function takes as arguments the number of flame length classes and the size of each flame length class (e.g., 1 foot), so the user has control over the table precision.

3. Error Handling

3.1 Return Codes and Error Messages

Most fireLib functions return a status code of `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure. The two exceptions are the fuel catalog creation functions `Fire_FuelCatalogCreate`() and `Fire_FuelCatalogCreateStandard`(), which return a catalog handle on success or `NULL` on failure.

In the event of a failure, all fireLib functions write an error message to the catalog's error message buffer. The programmer may access and display the message using the macro:

```
error = FuelCat_Error(catalog);
```

The error message remains intact until overwritten by a subsequent message. The return status of the most recent fireLib function is also available via the macro:

```
status = FuelCat_Status(catalog);
```

Error messages issued by each fireLib function are discussed in the Function Reference.

3.2 `catalog` Handle Validation

All fireLib functions also validate the `catalog` handle argument to ensure it is not `NULL` and that it contains a valid magic cookie. The validation is performed within the standard C `assert()` function, which prints a short (compiler-dependent) message to `stderr` and aborts the program with a core dump.

If you are satisfied that your program is not passing around invalid catalog handles, you may circumvent the `assert()` test by the standard C mechanism of defining the macro `NDEBUG`. This may slightly improve your application performance.

4. Accessing Object Properties Using Macros

fireLib functions return a success or failure code rather than computation results. Access to catalog, fuel model, and fuel particle properties, including all input variables, stored intermediates, and all fire behavior outputs, is available to the programmer via C macros. These macros are used like C function calls to access or update current object properties.

A property macro consists of the "`Fuel_`" prefix followed by a descriptive label and a 1, 2, or 3 arguments. The first argument is always the `catalog` handle. If two or more arguments, the second is always the fuel `modelNumber`. If three or more arguments, the third argument is either the fuel particle index, life class index, or moisture class index.

The following symbols are used to represent the property macro argument types:

Macro Arguments		
Argument	Description	Allowable Values
c	Fuel catalog handle	
m	Fuel model number	0 - maxModels
p	Fuel particle index	0 - maxParticles
x	Fuel moisture content by size class	FIRE_MCLASS_1HR FIRE_MCLASS_10HR FIRE_MCLASS_100HR FIRE_MCLASS_1000HR FIRE_MCLASS_HERB FIRE_MCLASS_WOOD
l	Fuel particle life class	FIRE_LIFE_DEAD FIRE_LIFE_LIVE

For example, the macro `Fuel_Load(c,m,p)` refers to the loading of fuel particle `p` in fuel model `m` of fuel catalog `c`.

The following tables document the property macro names, arguments, units of measure, and whether the property is input by the user (I) or derived by the function (D).

4.1 Fuel Catalog Property Macros

Macros for accessing fuel catalog properties require one argument, the `catalog`, which must be a valid FuelCatalogPtr handle. A common use of a fuel catalog macro is to get the current error message:

```
message = FuelCat_Error(catalog);
```

Fuel Catalog Property Macros			
Macro	Description	Units	I/D
FuelCat_Error(c)	Pointer to current error message buffer.	none	D
FuelCat_FlameArray(c)	Pointer to allocated flame length table. NULL if no flame length table.	none	I
FuelCat_FlameClasses(c)	Number of classes in flame length table. (Zero if no flame length table)	none	I
FuelCat_FlameStep(c)	Size of each flame length class.	feet	I
FuelCat_MagicCookie(c)	Fuel catalog magic cookie.	none	D
FuelCat_MaxModels(c)	Maximum allowable fuel model number	none	I
FuelCat_ModelArray(c)	Pointer to allocated fuel model object array.	none	D
FuelCat_Name(c)	Pointer to allocated fuel catalog name.	none	I
FuelCat_Status(c)	Most recent fireLib function return code: FIRE_STATUS_OK or FIRE_STATUS_ERROR	none	D

4.2 Fuel Model Property Macros

Macros for accessing fuel model properties require two arguments; `catalog`, and `modelNumber`. The `catalog` argument must be a valid FuelCatalogPtr handle and the `modelNumber` argument is a fuel model number. To access, for example, the the light logging slash (fuel model 11) fuel bed depth:

```
depth = Fuel_Depth(catalog,11);
```

Fuel Model Property Macros			
Macro	Description	Units	I/D
Fuel_CombustionFlag(c,m)	Equals 1 if Fire_FuelCombustion() has been run for this model. Equals 0 if fuel model has been updated.	[0..1]	D
Fuel_Depth(c,m)	Fuel bed depth	feet	I
Fuel_Desc(c,m)	Pointer to allocated fuel model description	none	I
Fuel_MaxParticles(c,m)	Maximum particles allowed	none	I
Fuel_Mext(c,m)	Extinction moisture content	lb water / lb fuel	I
Fuel_Model(c,m)	Model number	none	I
Fuel_ModelPtr(c,m)	Pointer to allocated fuel model object	none	D
Fuel_Name(c,m)	Pointer to allocated model name	none	I
Fuel_ParticleArray(c,m)	Pointer to allocated particle array	none	D
Fuel_Particles(c,m)	Number of defined particles	[0..maxParticles]	D
Fuel_SpreadAdjustment(c,m)	Spread rate adjustment factor	none	I

4.3 Fuel Particle Property Macros

Macros for accessing fuel particle properties require three arguments; `catalog`, `modelNumber`, and the particle index. The `catalog` argument must be a valid FuelCatalogPtr handle, the `modelNumber` argument is a fuel model number. Particle indexes start at 0 and correspond to the order in which `Fire_FuelParticleAdd`() was called. To access, for example, the light logging slash (fuel model 11) 10-hour fuel load:

```
load = Fuel_Load(catalog,11,1);
```

Fuel Particle Property Macros			
Macro	Description	Units	I/D
Fuel_AreaWtg(c,m,p)	Surface area derived wtg factor	none	D
Fuel_Density(c,m,p)	Fuel particle density	lbs fuel / cu ft fuel	I
Fuel_Heat(c,m,p)	Low heat of combustion	BTU / lb fuel	I
Fuel_Live(c,m,p)	Fuel life code: FUEL_LIFE_DEAD or FUEL_LIFE_LIVE	[0..1]	I
Fuel_Load(c,m,p)	Loading	lb fuel / sq ft bed	I
Fuel_Moisture(c,m,p)	Moisture content	lb water / lb fuel	I
Fuel_ParticlePtr(c,m,p)	Pointer to fuel particle object	none	D
Fuel_Savr(c,m,p)	Surface area-to-volume ratio	sq ft fuel / cu ft fuel	I
Fuel_SiEffective(c,m,p)	Effective silica content	lb silica / lb fuel	I
Fuel_SigmaFactor(c,m,p)	exp(-138./characteristic sa-vol)	sq ft fuel / cu ft fuel	D
Fuel_SiTotal(c,m,p)	Total silica content	lb silica / lb fuel	I
Fuel_SizeAreaWtg(c,m,p)	Size class surface area wtg factor	none	D
Fuel_SizeClass(c,m,p)	Surface area-to-volume size class 0 > 1200 sq ft / cu ft 1 > 192 2 > 96 3 > 48 4 >16 5 > 0	[0..5]	D
Fuel_SurfaceArea(c,m,p)	Total surface area	sq ft fuel / sq ft bed	D
Fuel_Type(c,m,p)	Fuel particle type code: FIRE_TYPE_DEAD FIRE_TYPE_HERB FIRE_TYPE_WOOD	[0..2]	I

4.4 Fuel Model Environment Property Macros

Macros for accessing fuel model environment properties require two arguments; `catalog` and `modelNumber`. The `catalog` argument must be a valid FuelCatalogPtr handle and the `modelNumber` argument is a fuel model number. To access, for example, the current aspect setting for the light logging slash model (fuel model 11):

```
aspect = Fuel_Aspect(catalog,11);
```

The macro to access environmental fuel moisture requires a third argument, `mclass`, which indicates which of four dead fuel classes or two live fuel classes to access. To access the live herbaceous fuel moisture content:

```
herbMoisture = Fuel_EnvMoisture(catalog,11,FIRE_MCLASS_HERB);
```

Fuel Model Environmental Property Macros			
Macro	Description	Units	I/D
Fuel_Aspect(c,m)	Terrain aspect	degrees clockwise from north	I
Fuel_EnvMoisture(c,m,x)	Environmental moisture content where x is one of the macro constants: FIRE_MCLASS_1HR (dead < 0.25") FIRE_MCLASS_10HR (dead < 1.0") FIRE_MCLASS_100HR (dead < 3.0") FIRE_MCLASS_1000HR (dead > 3.0") FIRE_MCLASS_HERB (live) FIRE_MCLASS_WOOD (live)	lb water / lb fuel	I
Fuel_Slope(c,m)	Terrain slope	rise / reach	I
Fuel_WindDir(c,m)	Direction of wind heading	degrees clockwise from north	I
Fuel_WindSpeed(c,m)	Wind speed	feet / min	D

4.5 Fuel Model Fire Behavior Property Macros

Macros for accessing fuel model fire behavior properties require two arguments; `catalog` and `modelNumber`. The `catalog` argument must be a valid FuelCatalogPtr handle and the `modelNumber` argument is a fuel model number. To access, for example, the spread rate at the current azimuth for the light logging slash model (fuel model 11):

```
spread = Fuel_SpreadAny(catalog,11);
```

Fuel Model Fire Behavior Property Macros			
Macro	Description	Units	I/D
Fuel_AzimuthAny(c,m)	Current fire spread azimuth	degrees clockwise from north	I
Fuel_AzimuthMax(c,m)	Azimuth of maximum fire spread	degrees clockwise from north	D
Fuel_ByramsIntensity(c,m)	Fireline intensity at Fuel_AzimuthAny	BTU / ft fireline / sec	D
Fuel_BulkDensity(c,m)	Fuel bed bulk density	lb fuel / cu ft fuel bed	D
Fuel_Eccentricity(c,m)	Fire ellipse eccentricity	none	D
Fuel_EffectiveWind(c,m)	Effective wind speed	ft / min	D
Fuel_FlameLength(c,m)	Flame length at Fuel_AzimuthAny	feet	D
Fuel_HeatPerUnitArea(c,m)	Heat per unit area	BTU/ sq ft fuel bed	D
Fuel_LiveMextFactor(c,m)	Live fuel extinction moisture content	lb water / lb fuel	D
Fuel_LwRatio(c,m)	Fire ellipse length-to-width ratio	none	D
Fuel_PhiEffWind(c,m)	Effective wind spread rate factor	none	D
Fuel_PhiSlope(c,m)	Slope effect spread rate factor	none	D
Fuel_PhiWind(c,m)	Wind effect spread rate factor	none	D
Fuel_ResidenceTime(c,m)	Fire residence time	minutes	D
Fuel_RxIntensity(c,m)	Fire reaction intensity	BTU / sq ft fuel bed / min	D
Fuel_ScorchHeight(c,m)	Scorch height	feet	D
Fuel_Spread0(c,m)	No-wind no-slope spread rate	feet / min	D
Fuel_SpreadAny(c,m)	Spread rate at Fuel_AzimuthAny	feet / min	D
Fuel_SpreadMax(c,m)	Spread rate at Fuel_AzimuthMax	feet / min	D
Fuel_WindLimit(c,m)	1 if wind exceeds effective limit	[0..1]	D

5. Functional Review

There are eight functions for creating, querying, and destroying fuel catalog, fuel model, fuel particle, and flame length table objects, and four functions for determining fire behavior for fuel models.

5.1 Creating and Destroying Fuel Catalogs

```
FuelCatalogPtr
Fire_FuelCatalogCreate (
     char  *name,
     size_t maxModels )
```

Creates a fuel catalog instance and returns its handle. The new catalog has space for fuel models numbered 0 through `maxModels,` but none are defined until subsequent calls by `Fire_FuelModelCreate().` The function returns `NULL` if unable to create the catalog.

```
FuelCatalogPtr
Fire_FuelCatalogCreateStandard (
     char  *name,
     size_t maxModels )
```

Creates a fuel catalog instance, defines the 13 standard fire behavior fuel models (models 1-13) and a no-fuel model (model 0), and returns the new catalog's handle. Additional fuel models (14 - `maxModels`) may be defined using `Fire_FuelModelCreate()`. The function returns `NULL` if unable to create the catalog.

```
int
Fire_FuelCatalogDestroy ( FuelCatalogPtr catalog )
```

Destroys the specified fuel `catalog` and releases all memory allocated to the catalog's fuel models and fuel particles. The catalog handle is no longer valid. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

5.2 Creating and Destroying Fuel Models

```
int
Fire_FuelModelCreate (
     FuelCatalogPtr catalog,
     size_t modelNumber,
     char  *name,
     char  *desc,
     double depth,
     double mext,
     double adjust,
     size_t maxParticles )
```

Adds a new fuel `modelNumber` to `catalog`. If a fuel model with `modelNumber` already exists in the catalog, it is first destroyed before the new one is created. While space is set aside for up to `maxParticle` fuel particles, no particles are defined until subsequent calls to `Fire_FuelParticleAdd()`. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

```
int
Fire_FuelModelExists (
     FuelCatalogPtr catalog,
     size_t         modelNumber )
```

This function returns 1 if fuel `modelNumber` is currently defined in `catalog`, otherwise it returns 0.

```
int
Fire_FuelModelDestroy (
     FuelCatalogPtr catalog,
     size_t         modelNumber )
```

Removes `fuel modelNumber` from `catalog` and releases all memory allocated to the model and its fuel particles. The `modelNumber` may be re-used by a subsequent call to `Fire_FuelModelCreate()`. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

5.3 Creating Fuel Particles

```
int
Fire_FuelParticleAdd (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     size_t         type,
     double         load,
     double         savr,
     double         dens,
     double         heat,
     double         stot,
     double         seff )
```

Adds a fuel particle to fuel `modelNumber` in `catalog`. Once added, the only way to remove or redefine the fuel particle (except by destroying and redefining the entire fuel model) is by using fuel particle property macros (see section 4). The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

5.4 Creating and Destroying Flame Length Tables

```
int
Fire_FlameLengthTable (
     FuelCatalogPtr catalog,
     size_t         flameClasses,
     double         flameStep )
```

Creates a flame length look-up table for `catalog` and returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure. If no table is defined for `catalog`, `Fire_SpreadAtAzimuth()` and `Fire_FlameScorch`() calculate flame lengths using the BEHAVE equations. In some simulations, these calculations may consume most of the computation time, and using a look-up table is significantly faster.

This function creates a table with `flameClasses` categories (beginning at flame length 0) of size `flameStep` feet. The table is then populated and automatically used in all subsequent calls to `Fire_SpreadAtAzimuth`() and `Fire_FlameScorch`().

The flame length table may be redefined at any time by issuing another call to `Fire_FlameLengthTable()` with new `flameClasses` and `flameStep` values. The flame length table is destroyed by calling `Fire_FlameLengthTable()` with a 0 value for `fireClasses`, after which all flame lengths are again calculated rather than looked up.

5.5 Calculating Fire Behavior

```
int
Fire_FuelCombustion (
     FuelCatalogPtr catalog,
     size_t         modelNumber )
```

Calculates all intermediate fuel bed and combustion variables that are solely dependent upon fuel bed characteristics (stage 1). It is normally not called directly by the user as it is automatically called by `Fire_SpreadNoWindNoSlope()` whenever fuel `modelNumber` has been defined or updated. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

A large number of intermediate fuel and combustion variables are calculated and stored by this function. The original BEHAVE System spends approximately 80% of its computation time in the redundant calculation of these non-variant values. Isolating these computations within a function which is called only once per fuel model dramatically improves performance.

```
int
Fire_SpreadNoWindNoSlope (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     double         moisture[FIRE_MCLASSES] )
```

Calculates reaction intensity, heat per unit area, and no-wind no-slope fire spread rate for fuel `modelNumber`. It automatically calls `Fire_FuelCombustion()` if needed for `modelNumber` . The `moisture[]` array contains the current fuel moistures (fraction of oven-dry weight) for four size classes of dead fuel (1-, 10-, 100-, and 1000-hour time lag) and 2 classes of live fuel (herbaceous and stem). The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

```
int
Fire_SpreadWindSlopeMax (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     double         windFpm,
     double         windDeg,
     double         slope,
     double         aspect )
```

Calculates the maximum spread rate and direction of maximum spread given the wind and terrain conditions. Once `Fire_SpreadNoWindNoSlope()` has been called to establish initial conditions for this fuel `modelNumber`, `Fire_SpreadWindSlopeMax()` may be called repeatedly with different input arguments to optimize simulation performance. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

```
int
Fire_SpreadAtAzimuth (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     double         azimuth,
     size_t         whichOutputs )
```

Calculates spread rate along the requested compass `azimuth` for fuel `modelNumber`, and optionally calculates Byram's fireline intensity, flame length, and/or scorch height along `azimuth` as requested by `whichOutputs`. Once `Fire_SpreadWindSlopeMax`() has been called to establish initial conditions, `Fire_SpreadAtAzimuth`() may be called repeatedly to get fire behavior at multiple azimuths. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

`whichOutputs` is specified by OR'ing together any of the following macro constants: `FIRE_NONE`, `FIRE_BYRAMS`, `FIRE_FLAME`, and/or `FIRE_SCORCH.` If `FIRE_NONE` is specified alone, only the spread rate is determined for the azimuth. `FIRE_NONE` and `FIRE_BYRAMS` require little computation time. `FIRE_FLAME` adds a call to the C `pow`() function (unless a flame length table is in use), and `FIRE_SCORCH` adds a call to both `pow`() and `sqrt`(), and therefore exacts the greatest performance pendalty.

If a flame length table has been defined for `catalog` via a previous call to `Fire_FlameLengthTable`(), it is used to look up the flame length. Otherwise, flame lengths are calculated directly.

```
int
Fire_FlameScorch (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     size_t         whichOutputs )
```

Calculates the flame length or scorch height along the azimuth established by the most recent call to `Fire_SpreadAtAzimuth()`. It is provided for those cases where it is faster to call `Fire_SpreadAtAzimuth()` with the `FIRE_NONE` option (perhaps to check all points of the compass), then calculate the flame length and/or scorch height for a specific azimuth. The function returns `FIRE_STATUS_OK` on success or `FIRE_STATUS_ERROR` on failure.

`whichOutputs` is specified by OR'ing together the macro constants `FIRE_FLAME` and/or `FIRE_SCORCH.`

If a flame length table has been defined for `catalog` via a previous call to `Fire_FlameLengthTable()`, it is used to look up the flame length. Otherwise, flame lengths are calculated directly.

6. Example: A Simple Fire Growth Simulator

A simple but functional fire growth simulator is presented as an example application using the fireLib API. The source code for this example resides in the file named `fireSim.c.`

Although the source listing is fairly short (about 200 lines of C code plus comments), it is capable of simulating fire growth within a spatially variable fuel, slope, aspect, wind speed, wind direction, and fuel moisture complex. It reads an ignition time map that may be seeded with any number of pre-planned ignitions in any shape and at any time or set of staggered times. The ignition map is modified as part of the simulation, and upon completion may be saved along with a flame length map to a GRASS format ASCII map file.

While the example simulation creates fuel, slope, aspect, and environmental conditions that are spatially constant, it is easily modified to create variable fuels, fuel breaks, variable weather, and multiple and time-staggered ignition points to acheive interesting effects. The figures below were generated by changing just a few source lines to acheive interesting effects.

The example remains simplistic, however, in that is does not allow temporal variability in wind speed, wind direction, or fuel moisture content; this is left as an exercise for the reader (I've always wanted to say that!).

Perhaps the best use of the example is to illustrate some common problems with cell-based contagious growth algorithms. The example simulation uses a simple 8-neighbor cell contagion algorithm for fire spread. These type of algorithms produce notoriously distorted fire perimeters whenever the direction of maximum spread approaches, but is not aligned with, a major neighbor vector (see French 1992 for a more thorough discussion).

Figure 1 maps the fire ignition times from a single ignition point in uniform fuel and environmental conditions without any slope or wind effect. It was created by changing wind speed to 0 (at `fireSim.c` line 42) and centering the ignition point (line 125). The octogonal shape is characteristic of the 8-neighbor contagion algorithm, and illustrates the amount of distortion from the expected circular perimeter.

Adding a 4 miles/hour windspeed (figure 2) shows how increasing spread rate produce fire shapes that do not resemble the expected elliptical geometry. It was generated by the source code listing in `fireSim.c.`

In figure 3 the wind direction was changed from 0 to 8 degrees east of north (line 43). This figure illustrates how a small deviation from an octogonal axis exaserbates the geometric distortion.

Figure 4 illustrates the ignition pattern resulting from two simultaneous ignition points. This was acheived by changing line 125 and inserting two new statements after line 26:

```
125  cell = Cols/3 + Cols*(Rows/4);
126a cell = 2*Cols/3 + Cols*(Rows/4);
126b ignMap[cell] = 1.0;
```

In figure 5 the two initial ignitions points by are staggered by100 minutes by changing line 126b:

```
126b ignMap[cell] = 100.0;
```

In figure 6 the fire burns around a fuel break spanning the middle 80% of the fuel array's central meridian. The fuel break was created by inserting the following statements after line 122:

```
122a
122b for ( cell=Cols*(Rows/2)+Cols/10, col=Cols/10;
     col<Cols-Cols-10; col++, cell++)
122c      fuelMap[cell] = 0;
```

The burning pattern in an array of randomly assigned fuels is shown in figure 7. This effect was accomplished by changing line 110:

```
110  fuelMap[cell] = rand()%14;
```

7. Validation and Performance

The fireLib outputs were compared with BEHAVE outputs over a wide range of input variables and were found to be in agreement within 1 part in 10000.

The example fire growth simulator listed in Appendix A was used to test fireLib performance on a variety of computer-operating system-compiler configurations.

Performance of fireLib Simulation Example			
Computer - Processor	Operating System	Compiler (options)	Seconds
Intel Pentium 100	UnixWare (SVR4)	UnixWare C (-g)	17
		UnixWare C (-O)	16
Sun SparcStation2 (sparc 4c)	Sun OS 4.1.4	acc (-g)	92
		acc (-O)	72
IBM RS6000 Power PC 601	AIX 3.5	cc (-g)	23
		cc (-O)	16

8. Availability and Licensing

8.1 Distribution

The fireLib distribution may be obtained by downloading the SEM home page at *www.montana.com/sem* or by FTP from *ftp.montana.com/sem*. Included in the distribution are:

`CHANGES`	documents any changes in fireLib distribution
`fireLib.c`	fireLib source code file
`fireLib.h`	fireLib header file
`fireLib.ps`	fireLib manual in Postscript (tm) format
`fireSim.c`	the example fire simulator source code
`license.txt`	fireLib license terms
`makefile`	`make` file to create the exdample `fireSim` program
`README`	latest information on fireLib installation and updates

Each version of the fireLib distribution is compressed into a single file and labeled with the version number. You have a choice of three compression schemes when downloading the distribution; gzipped tar Unix text or zipped DOS text. For example, firelib-1.0.0.tar.gz and firelib-1.0.0.zip both contain the initial distribution. The first version featuring bug fixes will appear as firelib-1.0.1.tar.gz and firelib-1.0.1.zip. The first version featuring new capabilities will appear as firelib-1.1.0.tar.gz and firelib-1.1.0.zip.

8.2 Bugs

Send bug reports to *cbevins@montana.com*. Please send a working example of the code that causes the error; I can only fix the error if I am able to duplicate it.

8.3 Licensing

The fireLib library is copyrighted © by Collin D. Bevins. The file `license.txt` contains the entire license agreement, and is reproduced below:

Collin D. Bevins
Systems for Environmental Management
315 South Fourth East
P.O. Box 8868
Missoula, MT 59807
cbevins@montana.com
(406) 549-7478 or (406) 728-7130

9. Selected References

Albini, Frank A. Estimating wildland fire behavior and effects. General Technical Report INT-30. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1976a. 92 p.

Albini, Frank A. Computer based models of wildland fire behavior: a user's manual. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1976b. 68 p.

Anderson, Hal E. Predicting wind-driven wildland fire size and shape. Research Paper INT-305. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1983. 26 p.

Andrews, Patricia L. BEHAVE: fire behavior prediction and fuel modeling system - BURN Subsystem, part 1. General Technical Report INT-194. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Research Station; 1986. 130 p.

Finney, Mark A. Modeling the spread and behavior of prescribed natural fires. In Proceedings of the 12th Conference on Fire and Forest Meteorology: 138-14; 1994.

.

Finney, Mark A. And Patricia L. Andrews The FARSITE fire area simulator: fire management applications and lessons of summer 1994. Interior West Fire Council Meeting and Symposium; Coeur d'Alene, ID; 1994.

French, Ian A. Visualisation techniques for the computer simulation of bushfires in two dimensions. M.S. Thesis, University of New South Wales, Australian Defense Force Academy; 1992.

Kourtz, Peter S and W. G. O"Reagan. A model for a small forest fire to simulate burned and burning area for use in a detection model. Forestry Science 17(2): 163-169.

Richards, G. D. An elliptical growth model of forest fire fronts and its numerical solution. Int. J. Numer. Meth. Eng. 30: 1163-1179; 1990.

Rothermel, Richard C. A mathematical model for predicting fire spread in wildland fuels. Research Paper INT-115. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1972. 40 p.

Rothermel, Richard C. How to predict the spread and intensity of forets and range fires. General Technical Report INT-143. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Research Station; 1983. 161 p.

Rothermel, Richard C. Predicting behavior and size of crown fires in the northern Rocky Mountains. Research Paper INT-438. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1991.

10. Function Reference

This section contains the technical reference to all fireLib functions. Each reference includes the following information:

Signature:

The ANSI standard C function declaration and required header files.

Arguments:

A description of each function argument and, when appropriate, its units of measure.

Returns:

The function return type and return codes.

Description:

A narrative of the function's purpose and operation.

Side Effects:

Lists all objects that are created or destroyed and all object properties that are updated.

Error Messages:

Lists error messages that may appear in the `FuelCat_Error` property if the function returns an error condition.

See Also:

Lists related functions.

Fire_FlameLengthTable()

Signature:

```
#include <fireLib.h>
int Fire_FlameLengthTable (
     FuelCatalogPtr catalog,
     size_t         flameClasses,
     double         flameStep )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()`or `Fire_FuelCatalogCreateStandard()`.
`flameClasses`	Number of classes within the table or 0 to delete an existing table.
`flameStep`	Size of each flame class in feet. The first flame class covers zero through `flameStep` feet.

Returns:

`FIRE_STATUS_OK` on success, `FIRE_STATUS_ERROR` on failure.

Description:

`Fire_FlameLengthTable()` creates a flame length look up table for `catalog`. In the absence of this table, `Fire_SpreadAtAzimuth()` and `Fire_FlameScorch()` calculate flame lengths using the BEHAVE equations, which use the `pow()` function and can consume most of the computation time in some applications. Defining a flame length table may improve performance by forcing these functions to look up rather than calculate flame length.

The flame length table is redefined by calls to `Fire_FlameLengthTable()` using new values for `flameClasses` and `flameStep`. Any existing flame length table is destroyed by calling this function with a `flameClasses` value of zero, after which all flame lengths are again calculated.

Side Effects:

Any existing flame length table is destroyed.
`FuelCat_FlameArray(c)` property is set to `NULL`.
`FuelCat_FlameClasses`(c) and `FuelCat_FlameStep(c)` properties set to zero.
If `flameClasses` is zero, returns `FIRE_STATUS_OK`.
Allocates a new flame length table pointed to by `FuelCat_FlameArray(c)` property.
`FuelCat_FlameClasses(c)` property set to `flameClasses`.

`FuelCat_FlameStep(c)` property set to `flameStep`.

Error Messages:

Fire_FlameLengthTable(): unable to allocate flame length table with `<flameClasses>` classes of `<flameStep>` feet.

Examples:

To create a flame length table with 200 1-foot flame length classes:

```
Fire_FlameLengthTable(catalog, 200, 1.0);
```

To destroy the above table and create a new table with 500 6-inch classes ranging from 0.5 through 250 feet:

```
Fire_FlameLengthTable(catalog, 500, 0.5);
```

To destroy any existing flame length table and force direct calculation of flame lengths:

```
Fire_FlameLengthTable(catalog, 0, 0.);
```

See Also:

`Fire_SpreadAtAzimuth(), Fire_FlameScorch().`

Fire_FlameScorch()

Signature:

```
#include <fireLib.h>
int Fire_FlameScorch (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     size_t         whichOutputs )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()`or `Fire_FuelCatalogCreateStandard()`.
`modelNumber`	Fuel model number.
`whichOutputs`	Flag indicating which fire behavior outputs to calculate. Specify this argument by OR'ing together `FIRE_FLAME` and/or `FIRE_SCORCH`.

Returns:

`FIRE_STATUS_OK` on success, `FIRE_STATUS_ERROR` on failure.

Description:

`Fire_FlameScorch()` determines the flame length and/or scorch height along the azimuth established by the most recent call to `Fire_SpreadAtAzimuth`(). It is provided for those cases where it is faster to call `Fire_SpreadAtAzimuth`() without flame length or scorch height computations, and then later ask for these values.

If a flame length table has been defined for `catalog` via a previous call to `Fire_FlameLengthTable`(), it is used to look up the flame length. Otherwise, flame lengths are calculated directly.

Side Effects:

`Fuel_FlameLength`(c,m) updated if `whichOutputs` includes `FIRE_FLAME`.
`Fuel_ScorchHeight`(c,m) updated if `whichOutputs` includes `FIRE_SCORCH`.

Error Messages:

Fire_FlameScorch(): fuel model <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

See Also:

`Fire_FlameLengthTable()`.

Fire_FuelCatalogCreate()

Signature:

```
#include <fireLib.h>
FuelCatalogPtr Fire_FuelCatalogCreate (
     char      *catalogName,
     size_t     maxModels )
```

Arguments:

`catalogName`	A unique name assigned to this fuel catalog.
`maxModels`	Maximum fuel model number allowed in this catalog.

Returns:

Returns a non-`NULL` handle (`FuelCatalogPtr`) on success, `NULL` on failure.

Description:

`Fire_FuelCatalogCreate()` creates a new fuel catalog object capable of holding `maxModels`+1 fuel models (e.g., fuel models numbered 0 through `maxModels`). The fuel catalog contains no fuel models until they are defined by subsequent calls to `Fire_FuelModelCreate()`.

Side Effects:

Allocates a new fuel catalog object referenced by the returned catalog handle.
`FuelCat_MagicCookie(c)` property initialized.
`FuelCat_Name(c)` property set to `catalogName`.
`FuelCat_MaxModels(c)` property set to `maxModels`+1.
Allocates a catalog error buffer pointed to by `FuelCat_Error(c)` property.
Allocates an array of fuel model object pointers pointed to by
    `FuelCat_ModelArray(c)` property.
`FuelCat_FlameArray(c)` property set to `NULL`.
`FuelCat_FlameClasses(c)` property initialized to zero.
`FuelCat_FlameStep(c)` property initialized to zero.
`FuelCat_Status(c)` property initialized to zero.

Error Messages:

All error messages are printed to `stderr`. Since a `NULL` return indicates that the fuel catalog object was not created, there is no error buffer for the catalog.

Fire_FuelCatalogCreate(): unable to allocate fuel catalog <`catalogName`> object.

Fire_FuelCatalogCreate(): unable toduplicate fuel catalog <`catalogName`> name.

Fire_FuelCatalogCreate(): unable to allocate fuel catalog <`catalogName`> error buffer.

Fire_FuelCatalogCreate(): unable to allocate fuel catalog <`catalogName`> with <`maxModels`> fuel models.

See Also:

`Fire_FuelCatalogCreateStandard`(), `Fire_FuelModelCreate`(), `Fire_FuelCatalogDestroy`().

Fire_FuelCatalogCreateStandard()

Signature:

```
#include <fireLib.h>
FuelCatalogPtr Fire_FuelCatalogCreateStandard (
     char       *catalogName,
     size_t      maxModels )
```

Arguments:

`catalogName`	A unique name assigned to this fuel catalog.
`maxModels`	Maximum fuel model number allowed in this catalog.

Returns:

Returns a non-`NULL` handle (`FuelCatalogPtr`) on success, `NULL` on failure.

Description:

`Fire_FuelCatalogCreateStandard()` creates a new fuel catalog object capable of holding `maxModels`+1 fuel models (e.g., fuel models numbered 0 through `maxModels`). If `maxModels` is less than 13, then space is allocated only for fuel models 0-13.

Fuel models 0-13 are then defined within the catalog. Fuel model 0 is a no-fuel model, while models 1-13 are the standard fire behavior models (Andrews 1976). Additional fuel models may be defined by subsequent calls to `Fire_FuelModelCreate()`.

Side Effects:

Produces the same side effects as the `Fire_FuelCatalogCreate`() function.
Fuel model 0 is defined as a fuel model with no fuel particles.
Fuel models 1-13 are defined for the NFFL (FMO) fire behavior models.

Error Messages:

All error messages are printed to `stderr`. Since a `NULL` return indicates that the fuel catalog object was not created, there is no error buffer for the catalog. May print error messages generated from within the `Fire_FuelCatalogCreate()`, `Fire_FuelModelCreate()`, or `Fire_FuelParticleAdd()` functions.

See Also:

`Fire_FuelCatalogCreate`(), `Fire_FuelCatalogDestroy`(), `Fire_FuelModelCreate`().

Fire_FuelCatalogDestroy()

Signature:

```
#include <fireLib.h>
int Fire_FuelCatalogDestroy (
     FuelCatalogPtr catalog )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate`() or `Fire_FuelCatalogCreateStandard`().

Returns:

Returns `FIRE_STATUS_OK`.

Description:

`Fire_FuelCatalogDestroy()` releases all memory allocated to the `catalog`, including its fuel models and fuel particles. The `catalog` handle is no longer valid.

Side Effects:

All fuel particle resources are released.
All fuel model resources are released.
All fuel catalog resources are released.

Error Messages:

None.

See Also:

`Fire_FuelCatalogCreate()`, `Fire_FuelCatalogCreateStandard`().

Fire_FuelCombustion()

Signature:

```
#include <fireLib.h>
int Fire_FuelCombustion (
     FuelCatalogPtr catalog,
     size_t         modelNumber )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()`or `Fire_FuelCatalogCreateStandard()`.
`modelNumber`	Fuel model number.

Returns:

`FIRE_STATUS_OK` on success, `FIRE_STATUS_ERROR` on failure.

Description:

`Fire_FuelCombustion()` calculates all the intermediate fuel bed and combustion variables that are solely dependent upon fuel bed characteristics. It is not normally called directly by the user, but rather by `Fire_SpreadNoWindNoSlope`() when the latter has detected a change in any fuel properties of fuel `modelNumber`.

Side Effects:

Calculates the following fuel bed and intermediate properties for `modelNumber`;

```
Fuel_AreaWtg(c,m,p)
     Fuel_SizeAreaWtg(c,m,p)
     Fuel_Moisture(c,m,p)
     Fuel_LifeAreaWtg(c,m,l)
     Fuel_LifeRxFactor(c,m,l)
     Fuel_BulkDensity(c,m)
     Fuel_ResidenceTime(c,m)
     Fuel_BulkDensity(c,m)
     Fuel_PropFlux(c,m)
     Fuel_SlopeK(c,m)
     Fuel_WindB(c,m)
     Fuel_WindK(c,m)
     Fuel_WindE(c,m)
     Fuel_FineDead(c,m)
     Fuel_LiveMextFactor(c,m).
```

Initializes the following fire behavior properties for `modelNumber` to zero; `Fuel_Spread0(c,m)`

```
     Fuel_RxIntensity(c,m)
     Fuel_HeatPerUnitArea(c,m)
     Fuel_SpreadMax(c,m)
     Fuel_AzimuthMax(c,m)
     Fuel_EffectiveWind(c,m)
     Fuel_PhiSlope(c,m)
     Fuel_PhiWind(c,m)
     Fuel_PhiEffWind(c,m)
     Fuel_Eccentricity(c,m)
     Fuel_WindLimit(c,m)
     Fuel_SpreadAny(c,m)
     Fuel_AzimuthAny(c,m)
     Fuel_ByramsIntensity(c,m)
     Fuel_FlameLength(c,m)
     Fuel_ScorchHeight(c,m).
```

`Fuel_LwRatio(c,m)` property initialized to 1.

Initializes the following environmental variables to zero;

```
     Fuel_WindSpeed(c,m)
     Fuel_WindDir(c,m)
     Fuel_Slope(c,m)
     Fuel_Aspect(c,m)
     Fuel_EnvMoisture(c,m,x).
```

`Fuel_CombustionFlag(c,m)` property set to 1.

Error Messages:

Fire_FuelCombustion(): fuel mode <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

See Also:

`Fire_FuelModelCreate`().

Fire_FuelModelCreate()

Signature:

```
#include <fireLib.h>
int Fire_FuelModelCreate (
     FuelCatalogPtr catalog,
     size_t          modelNumber,
     char           *modelName,
     char           *desc,
     double          depth,
     double          mext,
     double          adjust,
     size_t          maxParticles )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()or` `Fire_FuelCatalogCreateStandard().`
`modelNumber`	Fuel model number; must be less than or equal to the maximum number of fuel models allowed in `catalog`.
`modelName`	A short name for this fuel model (or `NULL`).
`desc`	A description for this fuel model (or `NULL`).
`depth`	Fuel bed depth (feet).
`mext`	Dead fuel extinction moisture content (lb water per lb fuel).
`adjust`	Spread rate adjustment factor (not used).
`maxParticles`	Maximum number of fuel particles that may be defined for this fuel model.

Returns:

`FIRE_STATUS_OK on success, FIRE_STATUS_ERROR on failure.`

Description:

`Fire_FuelModelCreate()` creates a new fuel model within `catalog`. If a fuel model already exists with this `modelNumber`, it is first destroyed before the new one is created, and no error message is generated. The fuel model contains no fuel particles until subsequent calls to `Fire_FuelParticleAdd`().

Side Effects:

Allocates a new fuel model object referenced by `catalog` and `modelNumber` and pointed to by `FuelCat_ModelPtr(c,m)` property.
`Fuel_Model(c,m)` property set to `modelNumber`.
`Fuel_Depth(c,m)` property set to `depth`.
`Fuel_Mext(c,m)` property set to `mext`.
`Fuel_SpreadAdjustment(c,m)` property set to `adjust`.
`Fuel_Name(c,m)` property points to `modelName` buffer.
`Fuel_Desc(c,m)` property points to `desc` buffer.
Allocates an array of fuel particle object pointers pointed to by the `Fuel_ParticleArray(c,m)` property.
`Fuel_MaxParticles(c,m)` property set to `maxParticles`.
Initializes all `Fuel_ParticlePtr(c,m,p)` properties to `NULL`.
`Fuel_CombustionFlag(c,m)` set to 0.

Error Messages:

Fire_FuelModelCreate(): fuel model <`modelName`> number <`modelNumber`> exceeds fuel catalog <`catalogName`> range [0..<`maxModels`>].

Fire_FuelModelCreate(): fuel model <`modelName`> number <`modelNumber`> depth <`depth`> is too small.

Fire_FuelModelCreate(): fuel model <`modelName`> number <`modelNumber`> extinction moisture <`mext`> is too small.

Fire_FuelModelCreate(): unable to allocate fuel model <`modelName`> number <`modelNumber`> for fuel catalog <`catalogName`>.

See Also:

`Fire_FuelModelExists()`, `Fire_FuelParticleAdd()`, `Fire_FuelModelDestroy()`.

Fire_FuelModelDestroy()

Signature:

```
#include <fireLib.h>
int Fire_FuelModelDestroy (
     FuelCatalogPtr catalog,
     size_t         modelNumber )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()or` `Fire_FuelCatalogCreateStandard()`.
`modelNumber`	Fuel model number.

Returns:

`FIRE_STATUS_OK on success, FIRE_STATUS_ERROR on failure.`

Description:

`Fire_FuelModelDestroy()` removes fuel `modelNumber` from `catalog` and releases all its resources including all its fuel particle resources.

Side Effects:

Releases all fuel particle resources for the `modelNumber`.
Releases all fuel model resources for the `modelNumber`.
`FuelCat_ModelPtr(c,m)` property for `modelNumber` set to `NULL`.

Error Messages:

Fire_FuelModelDestroy(): fuel model number <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

See Also:

`Fire_FuelModelCreate(), Fire_FuelModelExists().`

Fire_FuelModelExists()

Signature:

```
#include <fireLib.h>
int Fire_FuelModelExists (
     FuelCatalogPtr catalog,
     size_t         modelNumber )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()or` `Fire_FuelCatalogCreateStandard().`
`modelNumber`	Fuel model number.

Returns:

1 if `modelNumber` exists in `catalog`, 0 if it does not exist.

Description:

`Fire_FuelModelExist`() reports whether or not a fuel model object currently exists referenced by `catalog` and `modelNumber`. Fuel model objects are created by `Fire_FuelModelCreate`() and destroyed by `Fire_FuelModelDestroy`().

Side Effects:

None.

Error Messages:

None.

See Also:

`Fire_FuelModelCreate(), Fire_FuelModelDestroy().`

Fire_FuelParticleAdd()

Signature:

```
#include <fireLib.h>
int Fire_FuelParticleAdd (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     size_t         type,
     double         load,
     double         savr,
     double         dens,
     double         heat,
     double         stot,
     double         seff  )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()or` `Fire_FuelCatalogCreateStandard().`
`modelNumber`	Fuel model number.
`type`	Fuel particle type, must be one of the predefined macro constants `FIRE_TYPE_DEAD`, `FIRE_TYPE_HERB`, `or` `FIRE_TYPE_WOOD`.
`load`	Fuel particle load (pounds per square foot of fuel bed).
`savr`	Surface area-to-volume ratio (square feet of fuel particle surface area per cubic feet of fuel particle volume).
`dens`	Particle density (pounds of fuel particle per cubic feet of fuel particle).
`heat`	Low heat of combustion (BTU per pound of fuel particle).
`stot`	Total silica content (pounds of silica per pound of fuel particle).
`seff`	Effective silica content (pounds of silica per pound of fuel particle).

Returns:

`FIRE_STATUS_OK on success, FIRE_STATUS_ERROR on failure.`

Description:

`Fire_FuelParticleAdd()` creates a new fuel particle object and adds it to `modelNumber` in `cataolg`. The number of fuel particles a fuel model may contain is defined by the `maxParticles` argument to `Fire_FuelModelCreate`(). Each particle is assigned an index number as it is added to the fuel model; the first particle has index 0, the second has index 1, and so forth.

Side Effects:

Allocates a new fuel particle object pointed to by `Fuel_ParticlePtr(c,m,p)` property.
`Fuel_Type(c,m,p)` property set to `type`.
`Fuel_Load(c,m,p)` property set to `load`.
`Fuel_Savr(c,m,p)` property set to `savr`.
`Fuel_Dens(c,m,p)` property set to `dens`.
`Fuel_Heat(c,m,p)` property set to `heat`.
`Fuel_SiTotal(c,m,p)` property set to `stot`.
`Fuel_SiEffective(c,m,p)` property set to `seff`.
`Fuel_Live(c,m,p)` property set to `FIRE_LIFE_DEAD` or `FIRE_LIFE_LIVE`.
`Fuel_SurfaceArea(c,m,p)` property is calculated.
`Fuel_SigmaFactor(c,m,p)` property is calculated.
`Fuel_SizeClass(c,m,p)` property is calculated.
`Fuel_AreaWtg(c,m,p)` property is set to zero.
`Fuel_SizeAreaWtg(c,m,p)` property is set to zero.
`Fuel_Moisture(c,m,p)` property is set to zero.
`Fuel_Particles(c,m)` property is incremented.
`Fuel_CombustionFlag(c,m)` property set to zero.
`FuelCat_Status(c)` property is updated.

Error Messages:

Fire_FuelParticleAdd(): fuel model number <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

Fire_FuelParticleAdd(): fuel model <`modelNumber`> type value (arg #3) is not FIRE_TYPE_DEAD, FIRE_TYPE_HERB, or FIRE_TYPE_WOOD.

Fire_FuelParticleAdd(): unable to allocate fuel particle to fuel model <`modelNumber`> in fuel catalog <`catalogName`>.

See Also:

`Fire_FuelModelCreate(), Fire_FuelModelDestroy().`

Fire_SpreadAtAzimuth()

Signature:

```
#include <fireLib.h>
int Fire_FireSpreadAtAzimuth (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     double         azimuth,
     size_t         whichOutputs )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()`or `Fire_FuelCatalogCreateStandard()`.
`modelNumber`	Fuel model number.
`azimuth`	Compass azimuth for which spread rate and other fire behavior calculation is requested (degrees clockwise from north).
`whichOutputs`	Flag indicating which fire behavior outputs should be calculated. May be the macro constant `FIRE_NONE` or the OR'd values of `FIRE_BYRAMS`, `FIRE_FLAME`, and/or `FIRE_SCORCH`. The spread rate at `azimuth` is always calculated regardless of the value of `whichOutputs`.

Returns:

`FIRE_STATUS_OK on success, FIRE_STATUS_ERROR on failure.`

Description:

`Fire_SpreadAtAzimuth()` calculates the fire spread rate in the specified compass `azimuth`. It optionally calculates Byram's fireline intensity, flame length, and/or scorch height, depending upon the value of `whichOutputs`.

This function depends upon fuel model state conditions established by the most recent calls to `Fire_SpreadNoWindNoSlope()` and `Fire_SpreadWindSlopeMax()`.

Side Effects:

`Fuel_AzimuthAny(c,m)` property set to `azimuth`.
`Fuel_SpreadAny(c,m)` property is calculated.
`Fuel_ByramsIntensity(c,m)` property is calculated if `whichOutputs` includes the macro constant `FIRE_BYRAMS`.
`Fuel_FlameLength(c,m)` property is calculated if `whichOutputs` includes the macro constant `FIRE_FLAME`.
Fuel_ScorchHeight(c,m) property is calculated if `whichOutputs` includes the macro constant `FIRE_SCORCH`.
`FuelCat_Status(c)` property is updated.

Error Messages:

Fire_FireSpreadAtAzimuth(): fuel model number <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

See Also:

`Fire_SpreadNoWindNoSlope(), Fire_SpreadWindSlopeMax(), Fire_FlameScorch().`

Fire_SpreadNoWindNoSlope()

Signature:

```
#include <fireLib.h>
int Fire_SpreadNoWindNoSlope (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     double         moisture[FIRE_MCLASSES] )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()`or `Fire_FuelCatalogCreateStandard()`.
`modelNumber`	Fuel model number.
`moisture`	Array of `FIRE_MCLASSES` (6) fuel moisture contents (lbs water per lb of fuel). The array elements are referenced using the macro constants `FIRE_MCLASS_1HR`, `FIRE_MCLASS_10HR`, `FIRE_MCLASS_100HR`, `FIRE_MCLASS_1000HR`, `FIRE_MCLASS_HERB`, and `FIRE_MCLASS_WOOD`.

Returns:

`FIRE_STATUS_OK on success, FIRE_STATUS_ERROR on failure.`

Description:

`Fire_SpreadNoWindNoSlope()` calculates the fire reaction intensity, heat per unit area, and spread rate under no-slope and no-wind conditions. If the fuel bed or fuel combustion characteristics have not been calculated for `modelNumber`, or if `modelNumber` has had fuel particles added to it, this function first calls `Fire_FuelCombustion()`.

Side Effects:

Calls `Fire_FuelCombustion()` if `Fuel_CombustionFlag(c,m)` is zero.
`Fuel_ByramsIntensity(c,m)` property initialized to zero.
`Fuel_FlameLength(c,m)` property initialized to zero.
`Fuel_ScorchHeight(c,m)` property initialized to zero.
`Fuel_EnvMoisture(c,m,x)` property set to `moisture[]` contents.
`Fuel_Moisture(c,m,p)` properties are set to `moisture[]` contents.
`Fuel_RxIntensity(c,m)` property is calculated.
`Fuel_HeatPerUnitArea(c,m)` property is calculated.
`Fuel_Spread0(c,m)` property is calculated.
`Fuel_SpreadMax(c,m)` property initialized to `Fuel_Spread0(c,m)`.
`Fuel_AzimuthMax(c,m)` property initialized to zero.
`Fuel_SpreadAny(c,m)` property initialized to `Fuel_Spread0(c,m)`.

`Fuel_AzimuthAny(c,m)` property initialized to zero.
`FuelCat_Status(c)` property is updated.

Error Messages:

Fire_FireSpreadNoWindNoSlope(): fuel model number <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

See Also:

`Fire_SpreadAtAzimuth(), Fire_SpreadWindSlopeMax().`

Fire_SpreadWindSlopeMax()

Signature:

```
#include <fireLib.h>
int Fire_SpreadWindSlopeMax (
     FuelCatalogPtr catalog,
     size_t         modelNumber,
     double         windFpm,
     double         windDeg,
     double         slope,
     double         aspect )
```

Arguments:

`catalog`	Fuel catalog handle returned by `Fire_FuelCatalogCreate()`or `Fire_FuelCatalogCreateStandard()`.
`modelNumber`	Fuel model number.
`windFpm`	Wind speed in feet per minute.
`windDeg`	Wind bearing or heading in degrees clockwise from north.
`slope`	Terrain slope as fraction rise / reach.
`aspect`	Terrain aspect (azimuth of downslope direction) in degrees clockwise from north.

Returns:

`FIRE_STATUS_OK on success, FIRE_STATUS_ERROR on failure.`

Description:

`Fire_SpreadWindSlopeMax()` calculates the direction of maximum fire spread and the spread rate in the maximum spread direction. The results depend upon initial conditions established by the most recent call to `Fire_SpreadNoWindNoSlope()` for `modelNumber`.

Side Effects:

`Fuel_WindSpeed(c,m)` property set to `windFpm`.
`Fuel_WindDir(c,m)` property set to `windDeg`.
`Fuel_Slope(c,m)` property set to `slope`.
`Fuel_Aspect(c,m)` property set to `aspect`.
`Fuel_ByramsIntensity(c,m)` property initialized to zero.
`Fuel_FlameLength(c,m)` property initialized to zero.
`Fuel_ScorchHeight(c,m)` property initialized to zero.
`Fuel_PhiSlope(c,m)` property is calculated.

`Fuel_PhiWInd(c,m)` property is calculated.
`Fuel_PhiEffWind(c,m)` property is calculated.
`Fuel_EffectiveWind(c,m)` property is calculated.
`Fuel_WindLimit(c,m)` property is calculated.
`Fuel_LwRatio(c,m)` property is calculated.
`Fuel_Eccentricity(c,m)` property is calculated.
`Fuel_SpreadMax(c,m)` property is calculated.
`Fuel_AzimuthMax(c,m)` property is calculated.
`Fuel_SpreadAny(c,m)` property initialized to `Fuel_SpreadMax(c,m)`.
`Fuel_AzimuthAny(c,m)` property initialized to `Fuel_AzimuthMax(c,m)`.
`FuelCat_Status(c)` property is updated.

Error Messages:

Fire_FireSpreadWindSlopeMax(): fuel model number <`modelNumber`> doesn't exist in fuel catalog <`catalogName`>.

See Also:

`Fire_SpreadAtAzimuth(), Fire_SpreadNoWindNoSlope().`