AN ABSTRACT OF THE THESIS OF

David H. Graetz for the degree of Master of Science in Forest Resources presented on July 5, 2000. Title: The SafeD Model: Incorporating Episodic Disturbances and Heuristic Programming into Forest Management Planning for the Applegate River Watershed, Southwestern Oregon.

Abstract approved: Signature redacted for privacy.

K. Norman Johnson

A hybrid landscape optimization/simulation model called SafeD (Simulation and analysis of forests with episodic Disturbances) was built to address the needs of forest management planning in the Applegate River Watershed, southwestern Oregon (the Applegate Project).

There are two goals of the Applegate Project: 1) search for forest policies and practices that achieve goals set for the watershed; and 2) simulate forest condition over time (in the context of possible stochastic disturbances) considering the effects of different forest policies and practices.

The SafeD model implements a four-stage process to guide management of the forested landscape to achieve specified goals over a planning horizon (40 years). The first stage develops stand prescriptions, for each recognized forest stand type and condition, which are designed to achieve specific stand goals. The second stage selects the prescription for each stand. The selection of prescriptions is accomplished using a heuristic programming technique, called the Great Deluge Algorithm, which is designed to find the "optimal" prescriptions that satisfy goals at the landscape level. In stage three the episodic disturbance processes are initiated. The episodic disturbances includes fire and insect attacks with weather patterns providing the stochastic element. Fire is spread using the FARSITE fire spread model with fine-resolution landscape data (25 meters x 25 meters). Insect attacks occur during drought periods in stands with excess basal area. Stage four is the re-analysis and re-selection of stand prescriptions (for the remaining time in the planning period) to accommodate for disturbances in stage three.

A sample application of the SafeD model is presented here. Two landscape scenarios were developed. The first scenario contains two landscape goals. One goal is

to produce the greatest amount of big trees (>= 15" DBH) across the landscape. The second goal is actually a sub-watershed equivalent roaded acre (ERA) constraint. The second scenario presents a grow-only strategy to encapsulate the idea of leaving a landscape unmanaged. The results show several interesting conclusions which may have implications for forest management practices in the Applegate River Watershed. First, in order to maximize the number of big trees across the watershed timber harvesting will need to occur. Second, the effects of episodic insect disturbance negate the need for as much timber harvesting as would be projected without accounting for such disturbances. And third, fire plays a significantly less role, in regards to tree mortality, than insects will.

The SafeD Model: Incorporating Episodic Disturbances and Heuristic Programming into Forest Management Planning for the Applegate River Watershed, Southwestern Oregon

by

David H. Graetz

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented July 5, 2000
Commencement June 2001

Master of Science thesis of David H. Graetz presented on July 5, 2000.

APPROVED:

Signature redacted for privacy.

Major Professor, representing Forest Resources

Signature redacted for privacy.

Chair of Department of Forest Resources

Signature redacted for privacy.

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Signature redacted for privacy.

David H. Graetz, Author

ACKNOWLEDGEMENTS

There are a number of individuals to whom I am greatly indebted to for their role during these past three years which it has taken me to complete this thesis. I would like to acknowledge them in chronological order: Debbie Johnson for her willingness to first hire me at one job and then pass me along to the Applegate Project at the expense of that job; Norm Johnson for taking a chance with me, for giving me guidance and vision during the project, and allowing me to explore tasks I had no business exploring because he knew I wanted a challenge; John Sessions for his technical expertise, his comradeship, and his patience and understanding in answering my many questions; Bernie Bahro for his fire expertise, his California connections, and for his friendship; and Jim Agee for all his work in getting me various parameters for the project and his patience in explaining them to me.

I would also like to recognize and thank Kay and Tony Davenport for their never-ending supply of love and support. I know I'm getting too old to still be in school and complaining about money but those days are about over (hey, only three more years for my PhD.!).

My office-mates and peers who have put up with my gripes, solitude, and otherwise non-social behavior deserve credit: Jonathan Brooks and Andy Herstrom. My future advisor and mentor, Pete Bettinger, has been a great source of information and guidance which I hope will continue.

Finally, I certainly can't forget to acknowledge the three most important sources of inspiration in my life. I should list them in chronological order so I don't get myself in too much trouble: Kido, my faithful dog who sometimes forgets that and runs away, but I can understand it's in his blood and so I don't take it too personal; Dana, my faithful girlfriend who sometimes forgets that and – oh wait, that was Kido – her love and energy have pulled us this far and I hope that together we get further down the road of life; and Zorra, Dana's faithful dog, for being a constant source of amusement and being just so darn cute.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF APPENDIX TABLES

The SafeD model: Incorporating Episodic Disturbances and Heuristic Programming into Forest Management Planning in the Applegate River Watershed, Southwestern Oregon

INTRODUCTION

Introduction and Project Strategy

This thesis presents my contribution and research for the Applegate River Watershed Forest Simulation Project (hereafter called the Applegate Project). The objective of the Applegate Project is to develop a forest landscape simulation model to use in evaluating the potential effects of different policies and forest management practices over time to achieve goals for the forest of the Applegate River Watershed in the context of possible stochastic events. The resulting model has been named SafeD (Simulation and analysis of forests with episodic Disturbances).

A strategy was implemented to break the development of the SafeD model into more manageable pieces. Stand level and landscape level goals were identified and serve as the logical spatial scales which will be used in this paper. Stand level refers to working at the scale of an individual stand and includes decisions on how to identify stands, how to classify stands, how to grow stands, and how to harvest stands. The stand level work done for the Applegate Project was completed by other team members. Landscape level refers to working at the scale of the area of interest (the 493,000 acre Applegate River Watershed). Decisions made at the landscape level are more complex to describe and model but they are essential to study because they look at the interactions of stands between each other and their role in a larger spatial context. There is some overlap between the stand and landscape level work and I will highlight that which is necessary to understand the work I completed at the landscape level. See Wedin (1999) for a more thorough overview of the stand level work completed for the Applegate Project.

My contribution to the Applegate Project was initially structured to provide Geographic Information System (GIS) support. That was later expanded to include undertaking the modeling effort and the development of the SafeD model. This thesis paper will focus on the modeling effort and the SafeD model. The GIS component of the Applegate Project, while an important and critical component, became secondary in the work I completed for the project. To this end, the GIS work I completed for the project will only be mentioned briefly and as needed.

Science Team

The Applegate Project consists of a collaborative science team represented by various disciplines. Team members include: Dr. Jim Agee, Univ. of Washington, College of Forest Resources; Bernie Bahro, USFS R5, Fire and Fuels Specialist; Don and Ellen Goheen, USFS R6, Insect and Pathology Specialist; Dr. Norm Johnson, OSU, Dept. of Forest Resources; Debbie Johnson, OSU, Research Forest; Jim Kayser, Biometrician; Dr. Chris Maguire, OSU, Dept. of Forest Science; Dr. John Sessions, OSU, Dept. of Forest Engineering; Heidi Wedin, OSU, graduate student in Dept. of Forest Resources; and myself.

Study Area Conditions and Management Guidelines

The Applegate River watershed is located in southwestern Oregon (Figure 1-1). The watershed is approximately 493,000 acres and drains into the Rogue River. Forest Service (FS) and Bureau of Land Management (BLM) lands comprise nearly two-thirds of the ownership and contain almost 80% of the forested lands. The watershed's 325,000 acres of federal lands were designated an Adaptive Management Area (AMA) in 1994 with the signing of the Northwest Forest Plan (USDA FS and USDI BLM, 1994). The Applegate AMA includes several allocations within its boundaries (Figure 1-2): Matrix, Riparian Reserves, and two Late-Successional Reserves (LSR). The remaining one-third of the watershed is in private ownership (mostly non-industrial) and the Red Buttes

Wilderness (federal land not in the Applegate AMA). The private lands generally occupy the valleys and lower elevations.

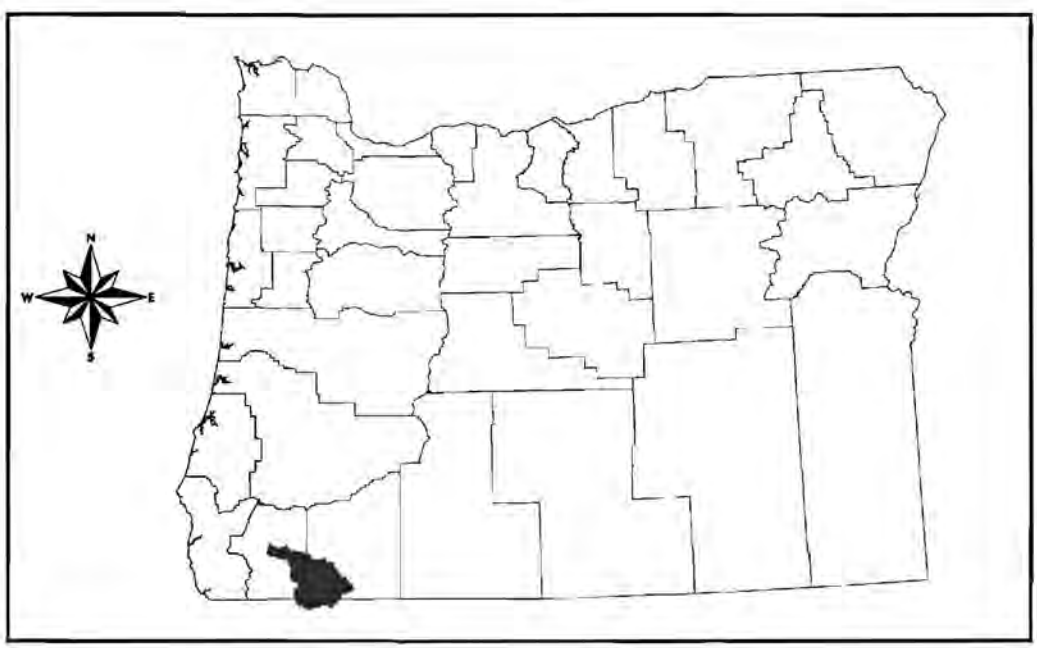

Figure 1-1: Applegate River Watershed locator

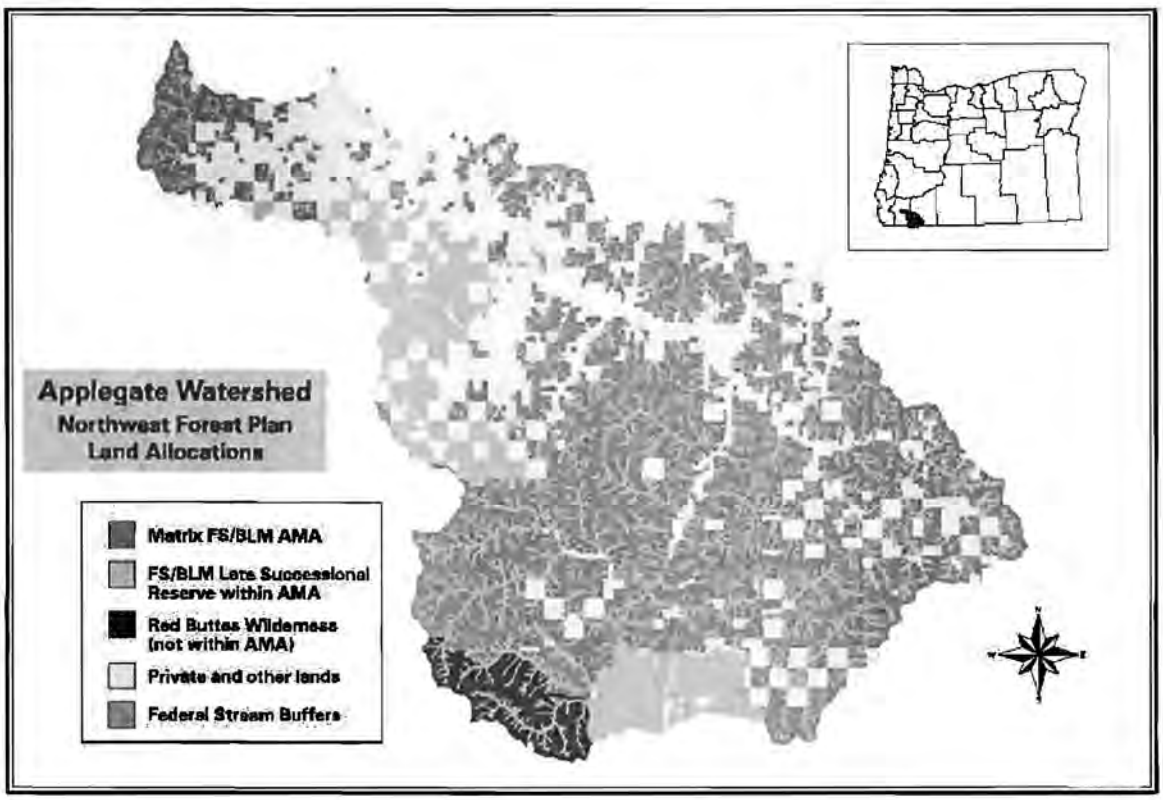

Figure 1-2: Northwest Forest Plan land allocations

An ecosystem health assessment of the Applegate AMA (USDI BLM and USDA FS, 1994) found the following:

- Increased risk of insect attack because of high stand densities.
- A younger and denser stand make-up than pre-European settlement.
- Stocking that exceeds carrying capacity over much of watershed.
- Increased risk of fire because of increased fuel-loading and stand conditions.

The BLM (USDI BLM and USDA FS, 1994), in a joint effort with other federal agencies within the watershed, sponsored the above ecosystem health assessment. The final report outlined some general management goals which included:

1. Reduce stand densities (of both merchantable and non-merchantable trees) and shrubs by thinning or prescribed fire.

2. Protect and restore riparian areas and late-successional habitat.

3. Increase the number of larger, older trees.

4. Promote, maintain, and restore shade intolerant species in designated Plant Association Groups (PAG's).

Additionally, the Standards and Guidelines for Management of Habitat for Late-Successional and Old-Growth Forest Related Species Within the Range of the Northern Spotted Owl (USDA FS and USDI BLM, 1994) provided for the production of wood commodities as a general management goal in the matrix allocation (but not necessarily an exclusive goal); harvest is also allowed in the LSR's to meet ecological objectives.

Problem Definition and Modeling Justification

The broad goal stated earlier to provide the Applegate Partnership with a model to simulate forest change over time and to find forest policies that would help the Applegate Partnership achieve its goals was very challenging. There are two distinct components within that goal: 1) search for forest policies and practices that achieve goals set for the watershed; and 2) simulate forest condition over time (in the context of possible stochastic disturbances) considering the effects of different forest policies and practices. Searching for forest policies and practices that achieve goals has its roots in classical forest planning; whereas simulating forest condition over time falls under the arena of classical landscape simulations. Both will be discussed in the Literature Review section. The Applegate Project attempts to bridge these two approaches by combining the spatial simulation of forest development on a large landscape, including stochastic disturbances, with the search for management actions that achieve multiple goals.

This problem involves the interaction of a variety of factors and processes, such as forest growth and yield, succession, management actions, and stochastic disturbances. There are various spatial and temporal scales at work which make the task more complex.

By defining these complex processes and their interactions logically and mathematically, landscape simulation models make it possible to examine assumptions about landscape change explicitly (Mladenoff and He, 1999). Perhaps Mladenoff and He (1999) stated the most compelling justification for landscape simulation models:

> ...modeling allows us to deduce results that otherwise cannot be investigated due to their complexity, such as landscape change over long time periods and the ecological ramifications of large disturbances, or diverse management regimes. (p. 125)

LITERATURE REVIEW

Introduction

The overall objective of the Applegate Project is to create a forest landscape simulation model to use in evaluating the potential effects of different policies and forest management practices over time to achieve goals for the forest of the Applegate River Watershed in the context of possible stochastic events. My ability to meet this objective requires a tool that can, 1) help evaluate the effects of different management policies and practices and 2) enable me to search for ways to achieve the goals set for the watershed. These policies and practices often span large temporal and spatial scales making assessment particularly complex. Simulation models are often the only way to assess such scenarios that cannot be tested under real-world conditions. Additionally, optimization methods are commonly used to determine policies and practices that achieve goals for landscapes. Thus, this literature review will first concentrate on the development of landscape simulation models and approaches, and particularly those that include stochastic elements (generally those which include natural disturbances). Then, methodologies used in forest management planning at the landscape level will be reviewed (landscape optimization). Lastly, I will review and characterize two landscape simulation models and a hybrid landscape simulation/optimization model.

It should be noted that anywhere the words "landscape simulation" or "simulation model" are seen in this thesis the intent is really "forest landscape simulation model". I may also use the terms "landscape optimization" or "optimization model"...again the intent is really "forested landscape optimization". This literature review and any discussion in this thesis are in reference to a forested landscape.

Landscape Analysis Modeling

I am using the term "landscape analysis" to encompass the idea of analyzing a large forested landscape for an assessment, strategic planning, or other purpose. The central theme of this literature review is to distinguish between "classical" landscape simulation models and "classical" landscape optimization models; both of which fall under

landscape analysis. I am also using the term classical to imply a traditional way or method of doing something. Figure 2-1 shows a hierarchical chart of these elements as I wish for them to be understood in the context of this literature review.

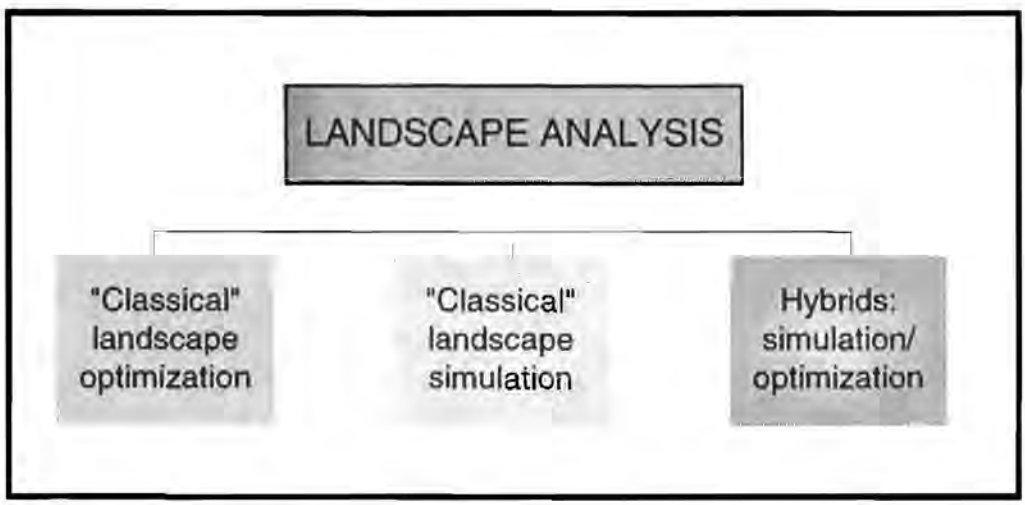

Figure 2-1: Hierarchical chart of landscape analysis

It is important to understand this structure because, as I will discuss, the SafeD model is a hybrid simulation/optimization model. Classical landscape simulation models typically show forest change over time as a function of known biological relationships and human activities that occur on the land. These relationships can be either spatial or non-spatial (distributional). Missing from classical simulation models is the ability to "search" for alternatives regarding what activities to place on the land (e.g. harvest, prescribe fire, hands-off) given a goal to reach. This ability to have a goal and search for ways to achieve that goal is encapsulated in classical forest optimization models. However, optimization models typically simplify biological relationships and do not address stochastic elements – everything must be programmed *a priori*. It is my goal in this literature review to demonstrate how these two approaches (simulation vs. optimization) differ and to suggest that the SafeD model is an improvement in landscape analysis modeling because it incorporates elements from both approaches.

In the course of this literature review I will compare and evaluate three antecedent simulation and optimization models. To accomplish this a framework for comparison

should first be established. I have chosen to look at two general components of any type of landscape analysis model: spatial data and desirable model characteristics.

Spatial Data and Desirable Model Characteristics

The "spatial data" components are those attributes that are related to the storage and use of spatial data. **Study area size** is used to help evaluate if models are working at the same scale. It may be unfair to compare a model that works on small landscapes (less than 1,000 acres) with one that works on large landscapes (over 100,000 acres). **Data structure** is the fundamental way in which the spatial data is stored and accessed for use. Vector (or polygon) format means that spatiality is maintained through lookup tables; raster format means that spatiality is inherent to the location of the data (as if on a grid). Related to the scale issue is that of **resolution.** By resolution I am referring to the size of the minimum mapping unit (MMU). In a raster model the MMU is often the size of each grid pixel (and the MMU size is uniform) whereas in a polygon model the MMU size may vary because polygons are seldom exactly the same size. By placing these attributes in the spatial data component I am not implying that they are unimportant – just that I have chosen not to evaluate them in a more critical manner as I will for the attributes in the desirable model characteristics component.

I have determined seven desirable model characteristics which I feel will allow me to evaluate whether or not a particular model would be suitable to address the needs of the Applegate Project. These seven characteristics and the need for each are:

1. **Recognize economical and ecological components**: The Applegate Partnership has indicated to the science team that they wish the model to have the ability to evaluate both these components. An economical component is necessary because there are real economic considerations the Partnership wishes to evaluate with different management scenarios. As well, the ecological component is necessary for the Partnership to assess the ecological effects of particular management scenarios.

2. **Optimize multiple goals**: The Applegate Partnership has indicated they wish to have the ability to set multiple landscape goals. Achievement of all goals is not necessarily a "hard goal"; but the ability to try and reach multiple goals and evaluate tradeoffs with different emphasis on goal-attainment is desired.
3. **Represent forest management activities**: Again, the Applegate Partnership has expressed the desire to evaluate the tradeoffs associated with active management, including timber harvest. Timber harvesting does occur within the Applegate Watershed and current regulations (both federal and state) allow for continuing harvesting.
4. **Represent stochastic elements**: Episodic disturbances are believed to play a major role in tree mortality within the watershed. The driving forces for these episodic disturbance events are stochastic in nature, for example drought. The ability to represent these stochastic elements will enhance any simulation model for the project.
5. **Represent fire – spatially explicit**: Fire has played a very important historical role in the Applegate Watershed. Neglecting fire and its effect would seriously skew and distort any simulation model for the watershed. *Spatially explicit* refers to the ability for a model to locate where an event occurs and allows for the spatial movement of events across the landscape. This is in contrast to a *distributional* approach in which events and/or their effects are spread on a "distributional" basis throughout some spatial unit.
6. **Represent insects – spatially explicit**: As with fire, mortality from insects has had a significant effect on the current forests of the Applegate Watershed. The current stand densities in the Applegate Watershed have high potential for future insect attacks. The ability to account for the occurrence and severity of insect attacks, in a spatially explicit manner, will greatly enhance any model applied to the watershed.
7. **Repeated simulations to assess variability**: Any stochastic landscape model should have the ability to run multiple times to assess variability. If a model was completely deterministic then multiple runs would be unnecessary; each run would be the same. On the other hand, stochastic models should have elements

that are different for each run. Having the ability to make multiple runs holding certain landscape parameters constant and allowing the stochastic elements to change gives a range of results that can form the basis for statistical analysis including variation and average conditions.

Classical Landscape Simulation and Modeling Approaches

Many landscape simulation models are considered to lie within an area of ecology called landscape ecology (Mladenoff and Baker, 1999). Landscape ecology can be broadly described as the study of ecological phenomena on large land areas. Golley (1993) suggested that landscape ecology, as applied in North America, derives its theoretical framework from ecosystem and community ecology, and its applied methodologies from environmental management. The development of classical landscape simulation modeling has its roots in forest ecology at a spatial scale and resolution fitting to the technology available at the time. Developments in this technology rapidly expanded starting in the early 1980's and correspondingly, so did landscape modeling. Mladenoff and Baker (1999) attribute two factors to the increasing scale and resolution of landscape modeling; availability of 30-meter resolution Landsat Thematic Mapper (TM) data and the rise of powerful small workstations with GIS software.

However, many of the simulation models developed in the early 1980's were still not spatially explicit and instead relied on distribution approaches (the distribution of land area among classes of landscape phenomena) (Mladenoff and Baker, 1999). This may be attributed to the lag time generally associated with the availability of new technology and the ability to use that technology. Kessells' (1979) gradient fire model was an exceptional model for the time in that it used spatially estimated vegetation and fuels data to simulate spatial fire patterns and post-fire succession.

A further development that helped propel landscape modeling was the introduction of mathematical and physical theories about properties of arrays of cells (Mladenoff and Baker, 1999). These theories and mathematical properties were around prior to the

1980's, but again, the proliferation of desktop computers gave landscape modelers the necessary tool to explore and use this information. Mladenoff and Baker (1999) discuss the properties of cellular automata and percolation modeling and how these components were incorporated into landscape simulation models.

Mladenoff and Baker (1999) give a concise review of early disturbance models which I will summarize here. Many of the initial fire disturbance models were developed to predict and understand fire behavior (for suppression) and were empirically based (Van Wagner, 1969; Rothermel, 1972). Fire disturbance models eventually made their way into forest management models (Kessell, 1979). Ecological research on forest disturbances were important for providing a framework for later ecological-based landscape simulation models. These disturbances include fire (Van Wagner, 1978; Johnson, 1992) and windthrow (Runkle, 1982; Frelich and Lorimer, 1991).

Classical Landscape Optimization

Early Forest Management and Planning Models

There has historically been a very close association between forest planning models and growth and yield models (Iverson and Alston, 1986). Mladenoff and Baker (1999) state that some distinction of scale can be made which differentiate larger scale, strategic planning or regional timber-supply models, and smaller scale, growth and yield models. One of the best known forest management models is FORPLAN of the US Forest Service (Iverson and Alston, 1986). However, early versions of models like FORPLAN were often too simple and were criticized for lacking ecological dynamics and variability, or spatial considerations (Johnson, 1992). Early models using Geographic Information Systems (GIS) were more of a decision-making software than models in a strict sense, and were not very ecologically driven (Mladenoff and Baker, 1999). As well, early GIS forest planning models typically did not consider natural disturbance rates or variability, or spatial interactions in their planning algorithms (Johnson and Scheurman, 1977; Hoganson and Burke, 1997).

Landscape Optimization Approaches

Strategic forest planning primarily has been focused on setting the level of timber harvest and the scheduling of timber harvest activities. Harvest levels based on controlling the volume harvested, the area cut, or both volume and area, have historically been calculated using simple formulas. Optimization models have recently become more pervasive in both private and public forest planning. Optimization models work on the principle that they attempt to maximize or minimize some quantity (usually called the *objective function* or *objective*) subject to reaching policy goals, and given certain choices for management that are allowed for individual parts of the forest or the entire forest. Policy goals are commonly formulated as constraints in an optimization model. Examples of objectives are to maximize timber harvest, maximize present net value (PNV), or minimize cost. Examples of policy goals are to maintain a non-declining yield of timber harvest over time, attain some distribution of acres among age-classes or seral stages, or to limit the rate of harvest in different portions of the forest (Davis and Johnson, 1987).

Policy goals are increasingly more complex and difficult to model. As a result, optimization models have been reformulated as "goal programs" (Sessions et al., 1999). In goal programming, constraints that were modeled as absolutes are transformed to allow for under- or over-achievement with an associated penalty value. The objective is generally to minimize the total penalty values. This formulation allows for recognition that it may be necessary in the short-term (or smaller spatial scale) to sacrifice and accept inferior values for some constraints in the course of achieving an overall better value in the long-term (or larger spatial scale). Additionally, it instills a sense of "fairness" to the model from often conflicting constraints given for various components of the model. For example, it might be desired to improve the habitat of species *X*, which is met by having eight snags per acre. At the same time, another goal may be to minimize the entire number of snags across the landscape. By allowing policy goals to be target values rather than absolutes, a solution might be found where six snags per acre can be created across the landscape and the overall number of snags is close to being minimized; neither goal was absolutely reached but each sacrificed a little in an acceptable compromise. Additionally, by adjusting the target values (or weights) an analyst is able to explore the

solution space. Policy goals are often modified as the tradeoffs surface from these adjustments.

Strategic planning systems based on forest-level optimization models have two main components: 1) the model formulation and 2) the solution technique (Sessions et al., 1999). I will discuss model formulation next and solution technique will be discussed later in the literature review because it plays a more significant role in my contribution to the Applegate Project.

Model Formulation

Model I and Model II:

Model I and Model II are terms used to label the two fundamentally different model formulations for optimizing forested landscapes (Johnson and Scheurman, 1977; Davis and Johnson, 1987). The main difference is in defining the decision variables for management activities and the way in which future (regenerated) stands are handled (Johnson and Scheurman, 1977). Model I defines decision variables that follow the life history of a stand over all planning periods. In Model II a stand may pass through several decision variables as stands are regenerated, grow, and die (Davis and Johnson, 1987).

A problem formulated as Model I can be formulated as Model II and vice versa. However, there are certain strengths and weaknesses of each that should be noted. The power of Model II comes from the ability to merge acres of like characteristics from across the planning area as they are regeneration harvested (Sessions et al., 1999). Fewer decision activities (thus fewer decision variables) are needed as acres are merged, but at the cost of losing some spatial definition in the management of future stands (Sessions et al., 1999). When such merging is not acceptable, Model I is usually a preferable formulation (Sessions et al., 1999).

Model III:

Boychuk and Martell (1996) used the term Model III to describe a generalized version of Model II whereby the stands pass through decision variables for reasons other than harvest, such as natural disturbances. An early example of Model III is seen in Reed and Enrico's model (1986) in which the expected burned area (from a wildfire) was subtracted from each age class in each time period, and added along with the cutover area to the youngest age class in the following period. Although Reed and Enrico (1986) described their model as stochastic, they actually used a "mean value" approach – the random proportion burned was replaced with its expected value (Boychuk and Martell, 1996). This approach has been found to have some problems. Hof et al. (1988) noted that attempting to use a mean value approach to a problem in a stochastic system leads to a high probability for infeasible solutions. Boychuk and Martell (1996) went on to compare the results of a stochastic programming problem (SPP) and the corresponding mean value problem when fire risk is considered in forest planning analysis. In the SPP formulation they represented stochastic fire loss by a discrete two-point probability distribution that yielded the desired mean and coefficient of variation. They compared only the first period solution and found that the mean value solution gave a good approximation to the SPP, but consistently over-harvested under some conditions (Boychuk and Martell, 1996).

Recent Landscape Analysis Models

Introduction

The threads of development from landscape ecology, disturbance models, forest management and planning models, new technology, and the cross-over of theories from mathematics have all interacted to arrive at the present state of landscape analysis modeling. There has also been a trend towards simulation models that are multi-scale and multi-process (Mladenoff and Baker, 1999). There are those with a narrow focus such as FARSITE (Finney, 1998) which uses grid-cell input data, a vector format to

model the spreading fire front, exogenous climate drivers that control fire spread, and a spotting routine that is stochastic and leapfrogs local dynamics.

Some models have incorporated non-spatial fire effects into forest planning approaches that seek management actions that achieve multiple goals (Reed and Enrico, 1986; Boychuk and Martell, 1996). Other models such as LANDIS (Mladenoff and He, 1999) have emphasized the ability to be spatially explicit while including stochastic elements and forest succession. And lastly, models such as SAFE FOREST (Sessions et al., 1999) have attempted to combine the spatial simulation of forest development on a large landscape, including wildfire disturbance and effects, with the search for management actions that achieve multiple goals. These multi-scale, multi-process models all have their groundwork based on earlier simulation models but interact in a spatially explicit format that is not simply neighborhood-based (Mladenoff and Baker, 1999).

Landscape Analysis Models

The SafeD model developed in this thesis is not without predecessors. Many simulation and optimization models have laid the groundwork for the work I completed. As I stated in the Literature Review introduction, it is my goal to distinguish between classical simulation models and classical optimization models and to suggest that a hybrid simulation/optimization model (such as SafeD) is an improvement . The two previous sections discussed classical simulation and optimization models. However, to further illustrate how a hybrid simulation/optimization model is an improvement in landscape analysis modeling I have chosen three antecedent analysis models to evaluate. The first two models, LANDIS and CLAMS, are forest simulation models – but not in a strict classical sense. The last model, SAFE FOREST, is a hybrid simulation/optimization model and is the most closely related to the SafeD model. I have chosen not to review any early simulation or optimization models that would be considered "classical" because landscape analysis modeling has progressed rapidly in recent years and I wish to evaluate models that are comparable to the SafeD model. This is not to dismiss their importance or role in landscape analysis modeling.

The LANDIS Model:

The LANDIS model by Mladenoff and He (1999) was designed to address the following needs:

1. Simulate large landscapes that are heterogeneous in terms of site conditions and initial vegetation conditions at the tree species level.
2. Simulate interaction of dominant forest disturbance regimes...fire, windthrow, and harvesting, with species-level forest succession.
3. Adapt to range of possible scales and map input-data of varied resolution.
4. Include spatially explicit ecological interaction, and mechanistic realism, while having modest input parameter needs. (p. 125)

The above needs are requirements for most forest landscape models and Mladenoff and He (1999) state they cannot all be optimized within a single model. The needs are framed by temporal and spatial scale, data availability, and parameter information for the area being modeled (Mladenoff and He, 1999).

The application of the model I reviewed was designed to look at how a regional landscape would recover from its current condition if natural successional processes operated, both with and without fire and wind disturbances (Mladenoff and He, 1999). The LANDIS model used a 10-year time step (over a 500 year planning horizon) to model a 3.7 million acre landscape in a transitional zone between boreal forest and temperate forest in northwestern Wisconsin (Mladenoff and He, 1999). A grid data structure was used with a grid-cell resolution of 200 meters x 200 meters.

Forest succession, seeding dynamics, and natural disturbances were the main components of the LANDIS model. LANDIS was designed as a tool to study species-level responses and changes in forest landscape pattern with varied natural and anthropogenic disturbances. These included stochastic fire and windthrow disturbances that moved on a cell-by-cell probability which the authors called spatially explicit (Mladenoff and He, 1999). In other words, the fire and windthrow events (once decided when and where they will happen or start) could move and spread across the landscape subject to behavior constraints and probabilities based on an individual cells' attributes.

Multiple simulations were run to assess the variability of conditions on the landscape, both with and without stochastic disturbances. With its ability to incorporate natural stochastic disturbances the LANDIS model "serves as a useful baseline against which to assess various landscape management or other change scenarios" (Mladenoff and He, 1999).

The CLAMS Model:

The Coastal Landscape Analysis and Modeling Study (CLAMS) project is a current effort to answer the question, "how [will] the current variety of land uses and forest policies in the [Oregon] Coast Range ... affect biological diversity, watershed processes, and economic and social outcomes" (Bettinger et al., 2000a). The CLAMS model incorporates strategic goals (aggregate harvest levels across large areas, multiple owners, and long periods) and tactical considerations (e.g., clearcut size limits, historical patch size distributions) (Bettinger et al., 2000a). The total study area is the Coast Range of Oregon (about 5 million acres) but the model runs independently for six "megasheds" just over 800,000 acres each.

The CLAMS model has its framework rooted in a raster data structure with Landsat TM data and digital elevation models (DEM) utilized in the preparation of model data input (Bettinger et al., 2000a). However, once the data is entered into the model it is used in a polygon format and spatial relationships are maintained through lookup tables. Stochastic disturbances such as fire, windthrow, drought, and insect outbreaks are currently not considered (Bettinger et al., 2000a).

The CLAMS model is a more detailed look at future conditions if current policies and practices were left in place. The level of spatial detail in the CLAMS model is remarkable for the size of the landscape being modeled.

The SAFE FOREST Model:

Authorized by Congressional funds in 1993 the Sierra Nevada Ecosystem Project (SNEP, 1996) was created. The primary goal for the SNEP team was:

> [to undertake] a scientific review of the remaining old growth in the national forests of the Sierra Nevada in California, and for a study of the entire Sierra Nevada ecosystem. (Johnson et al., 1998)

The project was an attempt to combine the spatial simulation of forest development on a large landscape, including wildfire disturbance and effects, with the search for management actions that achieve multiple goals. Many members of the scientific team for the Applegate Project were involved with the SNEP work and many ideas and methods have found their way from SNEP to the Applegate Project. For example, episodic disturbances, particularly fire, were identified as key elements in the shaping of the modern landscape within the Sierra. In fact, the effects and role of fire played such a large part in the development of the landscape simulation model that the model itself was dubbed "Simulation and Analysis of Fire Effects on FOREST" (SAFE FOREST) (Sessions et al., 1999).

The SAFE FOREST model used a vector data structure in which spatial relationships were kept through lookup tables. The main "modeling units" were called LSOG (Late Successional Old Growth) which were areas judged to be relatively uniform in type and distribution of vegetation patches (Johnson et al., 1998). The model was used on a 1 million acre landscape centered in the Eldorado National Forest and intermingled lands. Sessions et al. (1999) outlines a four-stage procedure for the SAFE FOREST model:

1. Find the set of activities (management actions) that best meets the goals for areas of late successional emphasis.
2. Find the set of activities that best meets the goals for the rest of forest.
3. Simulate the fires across the landscape for the planning periods based on randomly selected weather.
4. Adjust the schedule of activities, outputs, and effects following the fires. (p. 237)

Five goals were identified and specified in hierarchical fashion (high-to-low): 1) Increase the general extent and complexity of late-successional forests; 2) Reduce the potential for high-severity fire; 3) Restore riparian areas and watersheds; 4) Reintroduce historical ecosystem processes; and 5) Provide sustainable, cost-effective timber harvest

volume. The model search for management actions was designed such that achievement of a higher-order goal would not be compromised by attempts to achieve a lower-order goal (Sessions et al., 1999). Other considerations such as wildlife, silvicultural methods, and goal alternatives were also incorporated into the study (Johnson et al., 1998).

Following assignment of activities in stages one and two, fire was placed upon the landscape based on historical probabilities within each LSOG polygon (Sessions et al., 1999). Stochastic weather and wind patterns were used to determine fire spread.

The effects of fire were then estimated using the vegetation structure and composition at the time of the fire along with various topographic variables (Sessions et al., 1999). This is a partial spatial approach. The fire itself was not spread in a spatially explicit manner but the location of the fire within LSOG polygons and the use of probabilities allowed for a "spatial generalization" of the spread and extent of a fire. It should be noted that the SAFE FOREST partial spatial approach to spreading fire was a significant leap in incorporating fire disturbance processes into landscape analysis modeling. Multiple simulations were made to help assess the variability of wildfires through time (Sessions et al., 1999).

Summary of Antecedent Models

Table 2-1 shows a comparison matrix for the spatial data and desirable model characteristics of the three reviewed here. All three models recognize ecological components and represent forest management activities. Only the SAFE FOREST model has the ability to optimize multiple goals; which is an objective for the Applegate project. There is a great disparity in the ability to represent stochastic elements which is the other objective for the Applegate project. Both the LANDIS and SAFE FOREST model have stochastic elements, however they are limited to representing fire disturbance only. Both models ignore insects. The CLAMS model has no stochastic elements. Because there are no stochastic elements the CLAMS model has no need to undertake repeated simulations whereas the LANDIS and SAFE FOREST do.

		LANDIS	CLAMS	SAFE FOREST
Spatial Data Components	Study area size	1.5 million acres	5 million acres with 8 "megasheds"	1 million acres
	Data structure	raster	vector	vector
	Resolution (MMU)	200 meter x 200 meter	varies	varies
Desirable Model Characteristics Component	Recognize economical and ecological	ecological	both	both
	Optimize multiple goals	no	no	yes
	Represent forest management activities	yes	yes	yes
	Represent stochastic elements	yes	no	yes
	Represent FIRE - Spatially Explicit	yes	no	"partial"
	Represent INSECTS - Spatially Explicit	no	no	no
	Repeated simulations to assess variability	yes	no	yes

Table 2-1: Comparison matrix for recent landscape analysis models

Solution Techniques

Introduction

As I discussed earlier, there are two main components to strategic planning systems based on forest-level optimization: 1) the model formulation and 2) the solution technique (Sessions et al., 1999). Model formulation has been discussed. Solution technique refers to the particular mathematical technique used to solve a problem. I will discuss three broad classes of problem-solving techniques: linear programming, non-linear programming, and heuristics

Linear, Integer, and Mixed-Integer Programming

Linear programming is a class of problem-solving methods used for problems which are linear with respect to the relationships between the decision variables. Linear programming (LP) techniques can be used to find the mathematically optimal solution (Davis and Johnson, 1987). An optimal solution is the solution that gives the maximum or minimum value for the objective function given the constraints. Harvest scheduling models, such as FORPLAN (Johnson et al., 1980), rely on linear programming techniques. However, large landscape simulations frequently have constraints expressed in terms of maintaining a certain unit of area in a specific cover type and requires decisions variables to be binary (zero or one). Traditional linear programming of these types of problems is very difficult (Bettinger et al., 1997) and may require alternative techniques such as integer or mixed-integer programming.

Linear programming can find the optimal solution for a model whose variables are continuous. When decision variables are formulated as 0-1 variables the solution technique is called integer programming (IP) or mixed-integer programming (MIP) when both binary and continuous variables are present; both of which are considered extensions of LP (Hof and Joyce, 1992). For example, a harvest scheduling problem may be formulated such that the decision variables for a harvest unit are 0 (don't cut) or 1 (cut) for each period in the planning horizon. Often the spatial relationship between units is important and IP and MIP allow optimal solutions to be found when there are spatial

constraints (Hof et al., 1994). LP, IP, and MIP are useful techniques and many examples can be found of their application in forestry (Hof and Joyce, 1992; Hof et al., 1994; Davis and Johnson, 1987; Hoganson and Rose, 1984; Daust and Nelson, 1993; O'Hara et al., 1989; Clements et al., 1990). As with LP techniques, both IP and MIP techniques fail for problems where the decision variables are too many or are non-linearly related.

Nonlinear Programming

Linear, integer, and mixed-integer programming are suitable for problems where the decision variables have linear properties (i.e., plus or minus, + -). When operators other than these are present the problem is considered nonlinear (such as products, powers, and logarithms). Nonlinear problem solving techniques have practical size limitations on the decision variables and problems with convergence on local optimums. Currently a class of problem-solving methods called heuristics is being used to solve nonlinear forest management problems.

Heuristics

Zanakis and Evans (1981) trace the word heuristic from the Greek word "heuriskein" meaning "to discover". In landscape modeling the term heuristic is used to define a procedure to reduce search in problem-solving activities (Reeves, 1993). Reducing search in problem-solving is a goal in landscape modeling driven by limited computational capability and limited time; which can be related to the large number of choices generally associated with landscape problems. Because the ideal behind reducing search is to avoid looking at every possible problem solution (and thereby know the absolute optimal), a heuristic can only be considered to be "searching" for the optimal solution. This concept has led Reeves (1993 p6) to define heuristic as "a technique which seeks good (i.e. near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is". The utilization of heuristic programming

techniques may allow the integration of complex, and often non-linear relationships found in forest simulation models.

Taxonomy of Heuristic Methods:

There is a confusing array of references to heuristic techniques and classification in the published literature. There appears to be general consensus in the implementation of various heuristics but little recognition of an orderly schema of techniques. Perhaps that is because there are many variations and hybrids that defy orderly classification. In my search of forestry related literature regarding heuristics I have found a few discrete differences between techniques that will serve to bring some order (Figures 2-2 and 2-3).

To begin with, heuristics are a ***class*** of problem-solving methods. As discussed earlier other classes include linear programming, integer programming, and mixed-integer programming. The next level down are ***paradigms***. The three main paradigms found in forestry related heuristics are evolutionary algorithms, neighborhood search, and non-neighborhood search. These paradigms describes the fundamental difference in the search strategies. Evolutionary algorithms can be described as those whose search methods attempt to model natural selection and population genetics. Neighborhood search techniques employ a searching strategy that involves a defined "neighborhood" while non-neighborhood search strategies use some other search strategy.

A closer look at the neighborhood search paradigm reveals many variations and hybridizations on the theme of a neighborhood. I have chosen to illustrate one variation through a ***filter*** that describes the extent of the neighborhood (Figure 2-3). Extensive neighborhoods are those that involve looking at many other solution states before making a decision to move from one state to another. Simple neighborhoods usually only look at one or a few solution states before making a decision to move. Other filters that can be used in evaluating neighborhood searches (or even non-neighborhood searches and evolutionary algorithms) are whether or not a memory or acceptance criterion is employed. Memory is the idea that a solution remembers where it has been and using that knowledge in some decision-making process that determines how to move from one solution state to another. Acceptance criteria techniques are those that use an acceptance

criteria (or a single criterion) as the major factor for deciding the quality of a move. Filters are difficult to classify in a orderly fashion because most are used in a mix-and-match hybridization fashion that suits the modeler. For each of the three paradigms above there are many ***techniques*** which fall under each. Simulated annealing, the great deluge algorithm, and tabu search are three such techniques that have been used in forestry applications with success.

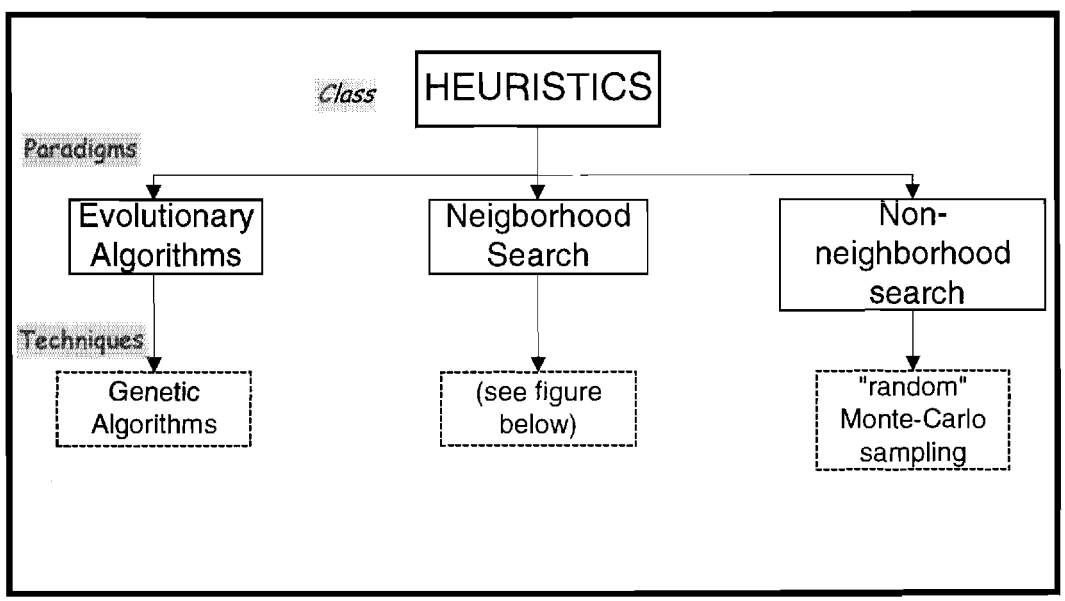

Figure 2-2: Hierarchical chart of heuristics

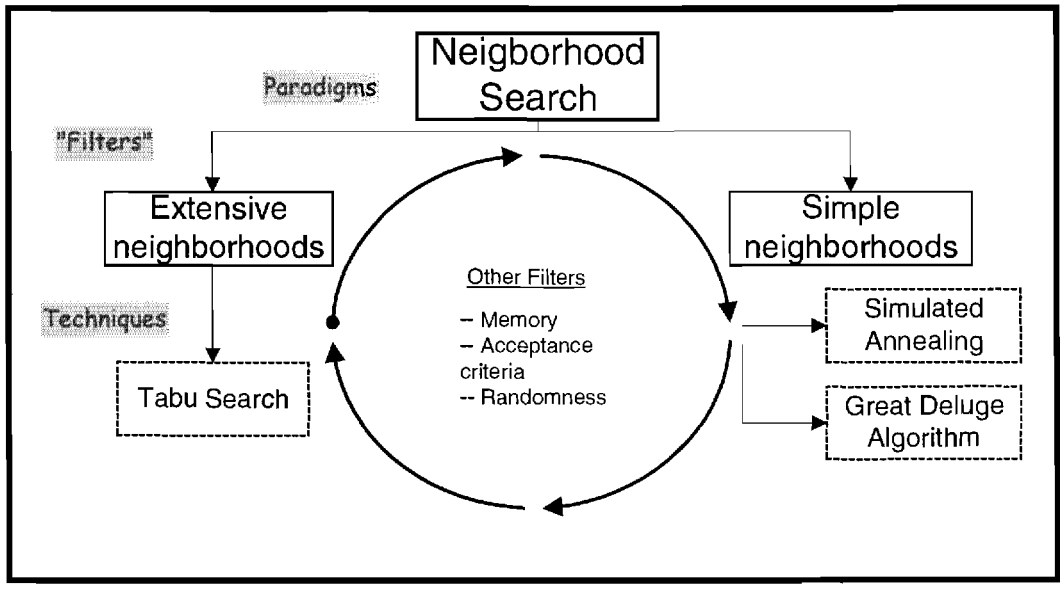

Figure 2-3: Sample classification of Neighborhood Search techniques

Simulated Annealing:

Annealing is a natural process in which the internal elements of a cooling body rearrange their order from a high-energy state to a low-energy state. In the high-energy state the elements of a system are molten and move freely. As the system is cooled, mobility is lost. If the system cools slowly (annealed), the elements crystallize into a stable state of minimal energy. If the system cools quickly (quenched), the elements harden into an unstable arrangement (Lockwood and Moore, 1993).

Simulated annealing (SA) is a heuristic programming technique that tries to mimic the annealing process described above. An SA algorithm tentatively alters the arrangement of elements in a system, evaluates the change in the objective function value, and then conditionally accepts or rejects the new arrangement (Dowsland, 1993). A "temperature" and "temperature reduction factor" are used in the SA process to evaluate the change (i.e., describe the energy state) in the objective function. New arrangements that improve the objective function are always accepted. Arrangements that worsen the objective function (analogous to adding energy to the system) are further evaluated by an additional acceptance criterion. In the early stages of the annealing process the acceptance criterion is less stringent and allows the system to accept

arrangements that worsen the objective function more frequently. As the algorithm reaches later stages the acceptance criterion becomes more stringent until, at some point, arrangements that worsen the objective function are no longer accepted (Dowsland, 1993). The acceptance criterion prevents the objective function from "greedily" converging on the closest local minima or maxima (Lockwood and Moore, 1993). Example of the use of SA for forestry applications can be found in Murray and Church (1995), Lockwood and Moore (1993), and Nelson and Liu (1994).

Great Deluge Algorithm:

The great deluge algorithm (GDA) is a recently developed variant on the above simulated annealing method for solving discrete combinatorial problems. The GDA was introduced by Gunter Dueck (1993) and proved superior to similar neighborhood search algorithms in solving a 442-city and 532-city Traveling Salesman Problem. The form of the GDA as presented by Dueck (1993) consisted of using a single parameter in the determination of whether or not to keep an inferior intermediate solution. The use of one parameter rather than two, as in a simulated annealing algorithm, is believed to de-sensitize the algorithm thus leading to equally good results even when parameter estimation and formulation is poor.

The GDA derives it name from the conceptual framework on which the algorithm works. If a problem were constructed such that the objective is to find the highest elevation in a fictitious country then the GDA would be one of maximization. The algorithm would start at some unknown location in the country and then it would "rain without end". The algorithm then walks around in this country trying to "keep its feet dry". However, the algorithm will tolerate water up to its ankles and so is allowed to walk in some inundated areas with the hope that there is dry land nearby. The water continues to rise and thus the dry land and acceptable ankle-deep water diminishes until the algorithm finally finds the "highest point" – determined by the fact that there is no more land left to walk around in without water going past the ankles.

Tabu Search:

Tabu search is a heuristic programming technique that employs a "memory" while aggressively exploring the solution space of an optimization problem. Exploration of the solution space is accomplished by making "moves" in the system, where a move is defined as the change in value of any one of the problem variables (Voß, 1993). A move may improve or diminish the quality of the objective function (same idea as a new arrangement as discussed in SA). A tabu search algorithm may contain a short- and long-term memory to help the algorithm intensify and diversify. Intensification is the ability to look around a particular area within a solution space without being confounded by non-improving moves (Voß, 1993). This is accomplished by a short-term memory list which restricts certain moves after the algorithm has made the move repeatedly. This prevents local cycling whereby the algorithm continuously finds a local minima or maxima (Glover and Laguna, 1993). Diversification is the ability to look in a completely new area of a solution space (Bettinger, 1998). This is accomplished by a long-term memory list which penalizes often-selected arrangements and forces the selection of new arrangements, which may allow the algorithm to search into unexplored regions of the solution space (Voß, 1993). Forestry applications of tabu search can be found in problems that address the scheduling of timber harvest subject to adjacency requirements (Murray and Church, 1995), and to harvest scheduling while meeting spatial goals for big game (Bettinger et al., 1997).

Summary and Conclusions

Strategic forest planning has primarily focused on setting the level of timber harvest and the scheduling of timber harvest activities. The SNEP and CLAMS projects are two studies that have included the traditional analysis and scheduling of timber harvest activities while also including the ability to model other policy goals, formulated as constraints. The LANDIS model is a spatially explicit and stochastic model that simulates forest landscape change over long time periods and over large, heterogeneous landscapes. The SAFE FOREST model also included stochastic wildfire but its spatial explicitness was limited to the size of its large modeling polygons.

As policy goals have become more complex, new programming techniques have been employed to solve landscape problems. Linear programming, integer programming, and mixed-integer programming are difficult to use for solving landscape-level spatial problems; either as a result of computational limitations or because of the complexity of the problem formulation (e.g., large number of choices or nonlinear operators) (Bettinger, 1998). Heuristic programming techniques have been found to be effective in solving optimization problems and their use in forestry applications is becoming more common (Lockwood and Moore, 1993; Murray and Church, 1995; Nelson and Liu, 1994). In particular, the Great Deluge Algorithm has been shown to be effective in evaluating a landscape problem with spatial constraints (Bettinger et al., 2000b).

With this literature review I hope to have demonstrated the three main drivers that have led me to develop the SafeD model. First is the need for a hybrid simulation/optimization model. The Applegate Partnership has expressed this need with two of their goals: 1) to simulate forest change over time and 2) to achieve goals set for the watershed. Second is the need to use heuristics solution techniques to solve our landscape problem. The number of decision variables associated with our problem and the need for spatial constraints make the use of traditional linear and non-linear programming problematic, if not impossible at this time. Finally, I have shown that an evaluation of recent landscape models reveals limitations in their ability to address the needs of the Applegate Project (see Table 2-1). The three models reviewed show at least one "deficiency" in the desirable model characteristics rows. Therefore, I have set out to build the SafeD model to both address the needs of the Applegate Partnership and to have all the desirable model characteristics shown in the matrix.

BROAD OBJECTIVE

The broad objective is to develop a landscape simulation model for the Applegate Partnership, land management agencies, and others to use in evaluating the potential effects of different policies and forest management practices to achieve goals set for the Applegate Watershed. These goals may include: 1) limiting insect and windthrow hazard, 2) limiting catastrophic fire hazard, 3) enhancing wildlife habitat, 4) improving fish habitat, and 5) providing for economic returns through timber harvest. In meeting these goals there may be a set of landscape sub-goals such as ensuring *X* number of snags are left on every 40 acres, maximizing even-flow of timber, maximizing present net value, maximizing wildlife habitat, or a combination of these. The actual goals used at the landscape level for the Applegate Project will be discussed in the Methods section of this thesis. There will be three main components of the overall simulation model (SafeD): 1) a stand prescription model (PREMO), 2) a disturbance model (which includes a fire model called FARSITE, an insect model, and a windthrow model), and 3) a landscape optimization model .

The above objective describes the final product desired (the SafeD model), however, there is a more subtle sub-objective that should be noted. Since the project will be done in a compartmentalized fashion the "bringing together" of all pieces will be an goal in itself. Management of the spatial data in a GIS, the stand prescription model, the fire model, the insect and windthrow models, and the landscape model all need to be integrated in the final step.

RESEARCH DESIGN

A four-stage process was developed to guide management of the landscape to achieve specified goals over a planning horizon (40 years). The first stage is the development of stand prescriptions, for each recognized forest stand type and condition, which are designed to achieve specific stand goals. The second stage is the selection and implementation of the prescription for each stand and to start the temporal "changing" of the landscape. The selection of prescriptions will be accomplished using heuristic programming techniques designed to find the "optimal" prescription that satisfies those goals at the landscape level – some of which have a spatial nature to them. In stage three the episodic disturbance processes are initiated. The episodic disturbances will include fire, insect attacks, and windthrow. Disturbance models will be stochastic and spatially explicit. Stage four is the re-analysis and re-selection of stand prescriptions (for the remaining time in the planning period) to accommodate for disturbances in stage three.

Results from the final landscape model can be compared to simulations of the landscape using other management approaches such as growth without active management or treatment of stands in accordance with current policies (based on ownership).

DATA AND COMPUTER RESOURCES

There are three broad classes of data that are utilized in the Applegate Project: 1) GIS spatial and tabular data, 2) stand and plot data, and 3) model-generated data. The spatial data originated in both raster and vector format but the SafeD model uses only raster data (thus vector data is converted through GIS to raster before use). This spatial data originated from a variety of sources with the main source being a database commissioned by the Applegate Partnership and put together by Interrain Pacific (now called EcoTrust), a Portland, Oregon company. The database is a compilation of spatial data within the Applegate River Watershed and I made only minimal efforts to verify the accuracy. See Table 5-1 for a description and brief explanation of the spatial data used.

DATA NAME[1]	SOURCE[2]	DESCRIPTION
Alloc	CD	Fed. land allocations (e.g. Matrix, LSR)
Aspect	CD	0-359 degrees
Cellid	Derived	Unique ID for every 25 meter cell
Elev	CD	In meters
Firehist	Derived	Areas of previous fire history (as known)
Minor	CD	6^{th}-field watersheds
Owner	CD	Land ownership
Pag	Derived	Plant Association Group
Stage	Derived	Initial structural stage
Slope	CD	In degrees
Strbuf	Derived	150-300 ft buffers around streams located on fed land
Treelist	Derived	Identifying lookup values to reference initial stand data
Veg	Derived	Initial vegetation classification

1. Data name is the name used in the SafeD model.
2. CD is the Applegate Watershed data produced by Interrain Pacific. Derived means that it was produced through GIS by either modifying data on the above CD or through unique rules and methods developed by the Applegate science team.

Table 5-1: Spatial data used for the Applegate Project

Associated with the spatial data are tabular data describing various attributes of the spatial data. The stand and plot data consist of tabular information that describes the biometric variables measured during plot exams by both the USDA Forest Service and

the USDI Bureau of Land Management (sources of original plot data). This data was originally handled by science team member Kayser. My project deals only with the ending "treelist" that result from an analysis of plot data. A treelist is a list of records for a stand in which every record represents some tree or portion of a tree (on a per acre basis) and all the necessary biometrics for that specific record (Table 5-2).

Items in a TREELIST								
Plot #	Status	TPA	Model Code	Report Code	DBH	Tree Height	Crown Ratio	Condition
Old plot # - not used by SafeD	0 = Snag 1 = Live 2 = Dwd		Unique code for species	Unique code for species				If "Status" is snag or dwd – code for condition

Table 5.2: Example of items and codes found in a treelist

Finally, when the model is run, we generate a number of intermediate data files. Some of these files will be needed later in the model, some will be needed for post-evaluation or analysis, and some can be discarded after use. In general, these intermediate data files are made in text format or stored in computer-coded arrays for later use.

A Gateway computer with a 18 GigaByte (GB) internal hard drive was purchased and used for the Applegate Project. The operating system is Windows NT, Version 4.01 running on a Pentium III Xeon 550 MHz CPU. All the model development, data storage, and data management was done on this machine. Both spatial and tabular data were stored on the hard drives and backed-up on a regular basis to a 4mm DAT tape. Incremental backups occurred daily and full backups were done on a weekly basis.

The GIS work was completed using ArcInfo v.7.2. The SafeD model itself was written in the C and C++ language (using Microsoft's Visual C++ v.5.0).

THE SAFED MODEL

General

The goal of the Applegate Project is to develop a landscape simulation model for the Applegate Partnership, land management agencies, and others, to use in evaluating the potential effects of different policies and forest management practices over time to achieve goals for the forest of the Applegate River Watershed in the context of possible stochastic events. The project is designed to incorporate guidelines set for AMA's. A key feature of AMA's is to:

> ... provide for development and demonstration of monitoring protocols and new approaches to land management that integrate economic and ecological objectives based on credible development programs and watershed and landscape analysis....[The guiding technical objective] is scientific and technical innovation and experimentation....[such as the] design and testing of effects of forest management activities at the landscape level. (USDA Forest Service and USDI Bureau of Land Management., 1994, p. D 2-4)

There are three main components to the overall simulation model - a stand prescription optimization model, a landscape optimization model, and disturbance models (which includes fire, insect, and windthrow models). The overall simulation model is the model I am describing in this thesis, the SafeD model (Simulation and analysis of forests with episodic Disturbances). As I will describe in more detail the three main components of the SafeD model are not necessarily embedded within the SafeD model code itself. In other words, there are external models that the SafeD model is dependent on. Handling the input and output of data from these external models is accomplished by the SafeD model. The SafeD model is also responsible for all the landscape optimization (to be discussed). The important point to remember is that the SafeD model accomplishes many processes within itself but is dependent upon other external models for some data and processes.

A four-stage process is used to test the impact of different management approaches on the landscape over the planning horizon (40 years). See Figures 6-1 and 6-2 for a flowchart of the following:

1. Stage one is the preparation of stand data and the development of stand prescriptions, for each recognized forest stand type and condition, which are keyed to different emphasis of the overall goals.
2. Stage two is the selection of a prescription for each stand based on the goals established for the watershed. The selection of prescriptions is accomplished using heuristic programming techniques designed to find a "near-optimal" spatial pattern that satisfies landscape goals.
3. Stage three is the initiation of the episodic disturbance processes. This includes a stochastic fire model, an insect model, and a windthrow model that are spatially explicit.
4. Stage four is the adjustment of the stand prescription to accommodate for the effects of disturbances in stage three and re-optimize at the stand and landscape level.

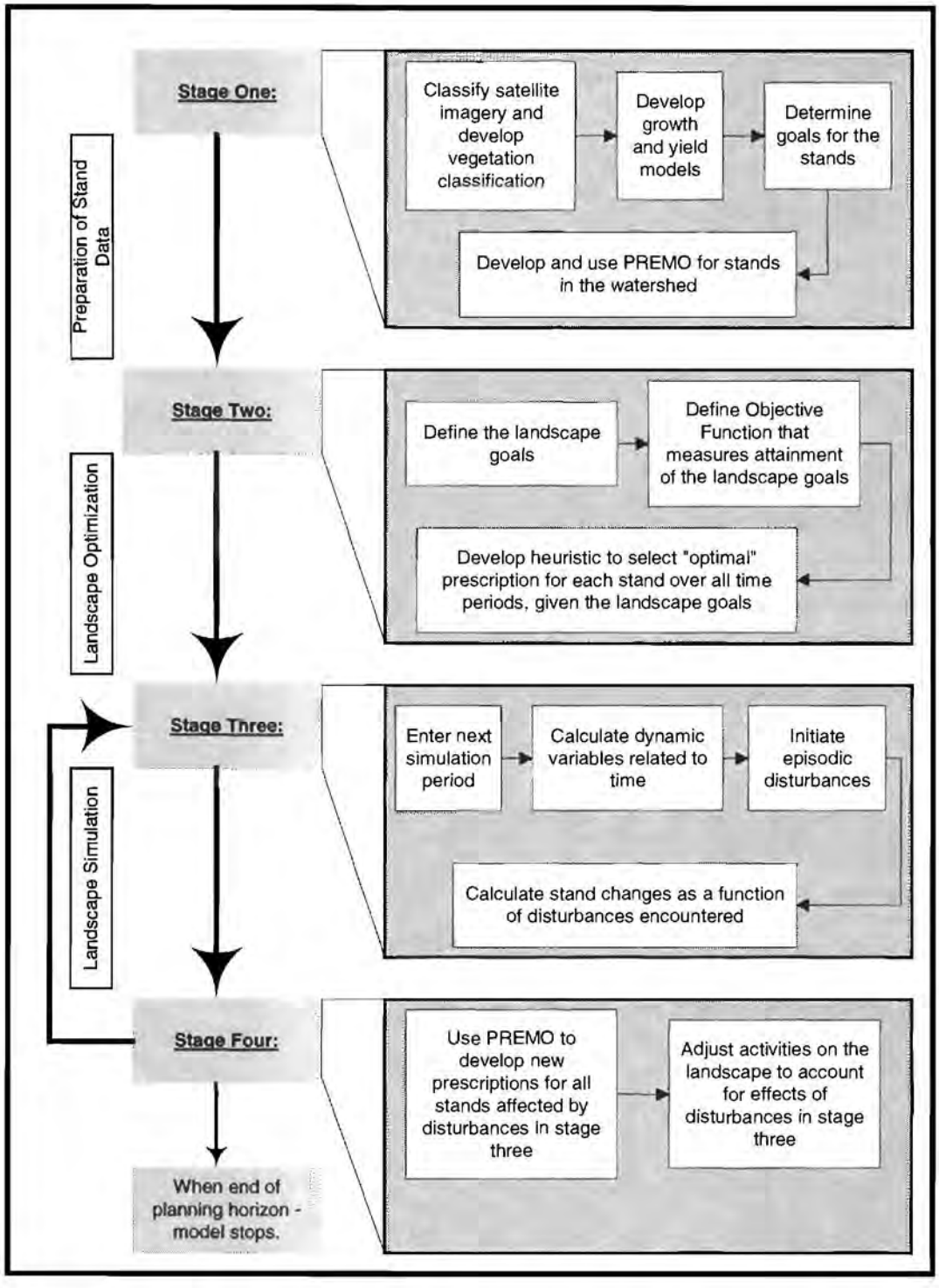

Figure 6-1: Flowchart of SafeD modeling stages

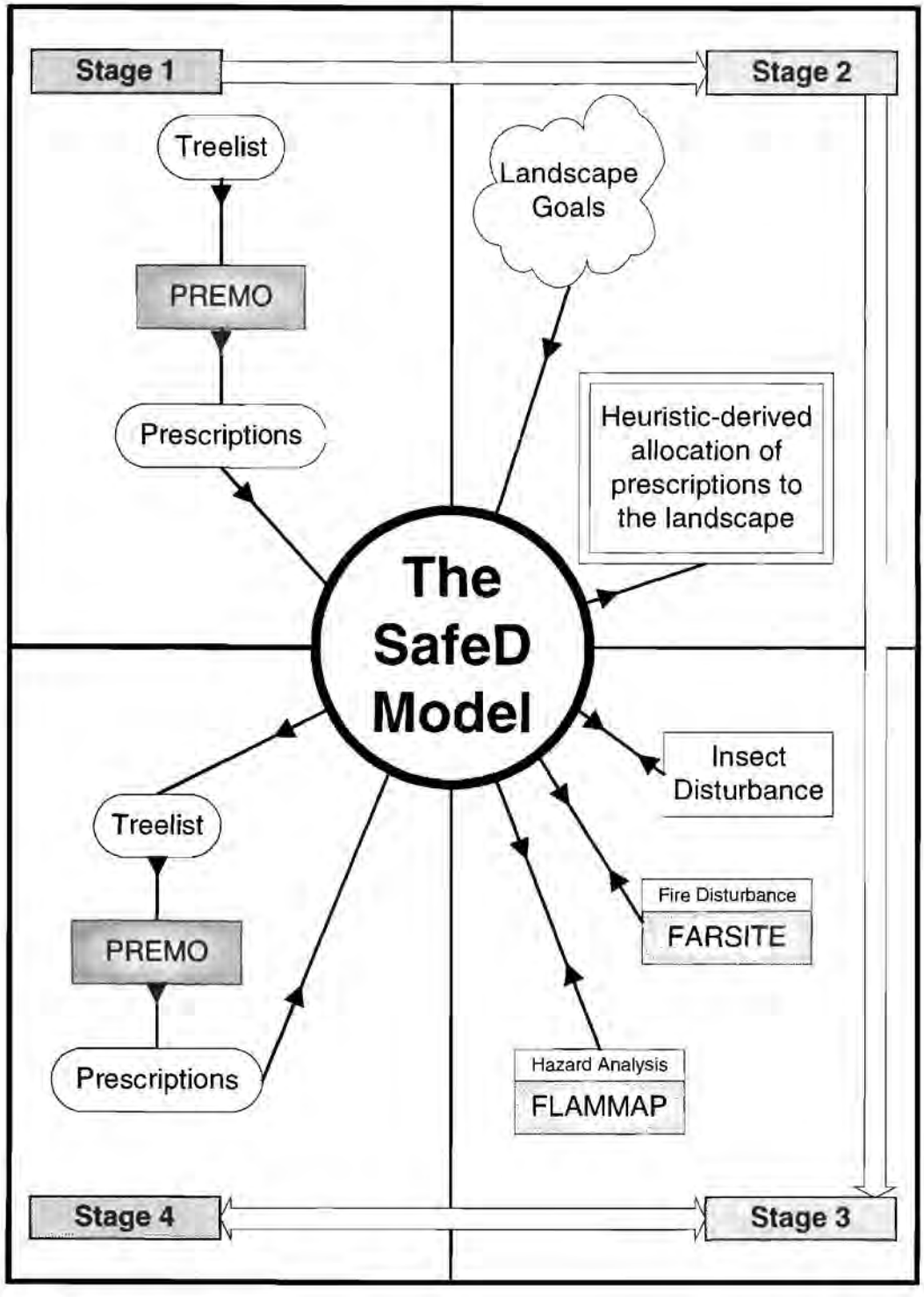

Figure 6-2: Interactions between major components of the SafeD model

Stage One

Vegetation classes and structural stages for the watershed have been developed using Landsat TM satellite imagery captured in August, 1993. Thirteen vegetation classes and fifteen structural stages are recognized within the watershed (Appendix A). Each combination of a vegetation class and structural stage could occur within three area-types identified on the landscape: normal, thinned, and fuel break. The three area-types were developed to address the fact that the Landsat image was taken in 1993 and activities have occurred on the landscape since that time. Data was obtained indicating that these activities included some thinning and fuel breaks. We were able to spatially identify many areas where a post-1993 thinning or fuel break was present.

Additionally, seven Plant Association Groups (PAGs) were developed by Agee (1999) to help determine the successional pathways. Each stand was assigned a PAG based on geology, elevation , precipitation, slope, and aspect. Appendix B shows the rules for PAG assignment:

An important component within the SafeD model is the identification of what is a "stand". The science team for the Applegate Project decided to break with traditional stand delineation (spatially defining polygons with similar vegetative or other attributes) and instead used each pixel from the classified Landsat TM imagery as the stand unit. That is, each 25m. X 25m. pixel is a stand in the SafeD model. "Stand type" refers to the unique combination of vegetation classification, structural stage classification, and area type. There are nine forested vegetation classes (and four non-forest classes), eleven forested structural stage classes (and four non-forest classes), and three area types. For the initial conditions in the Applegate Watershed there is a potential for 297 (9 * 11 * 3) unique stand types.

The first stage of the modeling process is the development of stand prescriptions that integrates growth, mortality, and achievement of stand goals. This is done for each of the unique stand types in the watershed. Project member Wedin (1999) developed the stand prescription optimization model. Wedin used the growth relationships found in the Forest Vegetation Simulator (FVS)(Dixon and Johnson, 1995) as the foundation for a new stand prescription generator called PREMO (PREscription generator under Multiple Objectives) (Wedin, 1999). Periodic insect, wind, and root disease mortality are

incorporated into PREMO. Relationships on decay of snags and down wood comes from Mellen and Ager's (1998) coarse woody debris model.

PREMO is designed to take stand data, represented by a list of live trees, dead trees, and down woody debris (called a treelist – see Data section), and create multiple prescriptions for the stand in response to emphases on goals that might be used to guide the management of the stand. A prescription, as I use in the context of the SafeD model, is a series of treelists (for a single stand type) that reflect the actual dynamic condition of the stand over the entire planning horizon. This is not the same as a "silvicultural prescription" which contains very specific information regarding how and when to treat a stand over the planning horizon. A prescription for the SafeD model is simply the ending series of treelist with the assumption that all the silvicultural activities (or none) have taken place and that growth and mortality is accounted for.

Five potential stand goals came out of discussions with the Applegate Partnership and others: 1) limiting fire hazard; 2) limiting insect and wind-throw hazard; 3) enhancing wildlife habitat; 4) improving fish habitat; and 5) providing a supply of timber in a cost-efficient manner. The tools for manipulating stand condition in order to achieve goals are growth, tree harvest, and snag creation. Measurements of goal attainment are based on stand variables as calculated by investigating the modeled residual tree list (standing live and dead trees, and down wood) or the modeled harvest treelist. Wedin uses the RLS-PATH optimization algorithm (Yoshimoto et al., 1990) to find prescription alternatives for each stand for each goal. More information on the stand prescription optimization model can be found in Wedin (1999).

The science team developed a unique approach to account for stand mortality by designing PREMO to function within the context of a larger landscape simulation model (i.e. the SafeD model). The mortality equations normally used by FVS were left out of PREMO and new periodic mortality equations were developed for PREMO to account for periodic insect, wind, and root disease. However, a larger percentage of stand mortality will occur episodically with the mortality occurring outside of PREMO (within the SafeD model). The SafeD model is designed to track stand mortality and thus can be used to evaluate whether or not the idea of representing stand mortality by episodic events is feasible.

The idea behind stage one and the use of PREMO is important enough to re-illustrate. Let's start with isolating a single stand type within the Applegate study area. Again, for this project the stands were identified as 25 meter pixels. Two or multiple adjacent pixels could be identical stand types but they are modeled and tracked as separate entities. Now assume that the stand selected is of **Red fir** vegetation class, falls into the structural stage class of **15" – 21" DBH with > 60% canopy closure**, and the area has not been modified since the 1993 Landsat image was taken (**normal** area type). These three components identify unique stand types within the watershed. There are many 25 meter pixels across the landscape that have these same components, and therefore, are considered as being identical stand types.

The Red fir stand type just described has a treelist associated with it that provides a description of the current stand in detail. We can manage that stand by allowing growth, tree harvest, and snag creation. The question is, "What do we do to the stand?" However, that can only be asked after answering the question, "What do you want the stand to look like at the end of the planning horizon?" And the latter question is what is used to identify a stand goal. If the stand goal is to create a stand that limits fire hazard, then what we are really saying is, "I want a stand that exists in such a condition that if a fire were to come through it, the adverse effects to the stand would be limited." Of course parameters and equations would need to be in place to measure the effects to the stand from the fire.

Now that we have identified a single stand goal for the Red fir stand type we can look at ways to treat the stand (or not treat the stand) to achieve the goal. And that is one of the functions of PREMO. PREMO tries to produce an "optimal" treatment (which results in a prescription) for a stand that takes into consideration the existing stand condition, the management activities available, and the desired ending stand condition. However, PREMO has to be pre-programmed with equations and functions for each desired stand goal. There are currently 19 stand goals available from PREMO:

1. Reduce fire risk
2. Reduce insect risk
3. Enhance fish habitat
4. Enhance wildlife habitat – complex structure
5. Enhance wildlife habitat – simple structure
6. Maximize PNV
7. Reduce fire and insect risk, and maximize PNV
8. Enhance fish and wildlife (complex) habitat, and maximize PNV
9. All goals with emphasis on maximize PNV
10. Grow only

11 – 19. Same as first nine except harvest not allowed until 3rd period (called Timing Choices)

In stage one we generate prescriptions for each unique stand type identified for all of the stand goals. Continuing with the Red fir example, PREMO will generate 19 different prescriptions for that stand type; each of which is optimized for a particular stand goal. The next question to ask is, "Which prescription do I actually select to implement?" and that is addressed in stage two. Again, see Wedin (1999) for a detailed explanation of how PREMO works in stage one.

The PREMO runs are made prior to the simulation runs. The SafeD model expects the prescriptions files to be available at run-time. The SafeD model is programmed to execute the PREMO runs in real-time as well (meaning PREMO is executed during the simulation run). This generally proves inefficient because multiple simulation runs are made and the PREMO runs are only needed once to create the prescriptions for the initial conditions (which do not change). If any changes to the PREMO code are made then new PREMO runs are made.

Stage Two

Introduction

Stage two is a three-phase process which completes the selection of specific prescriptions for each stand and is the core of the optimization part of the SafeD model. The prescriptions developed by PREMO in stage one are optimized at the stand level based on stand goals. The idea of stage two is to optimize the selection of specific stand

prescriptions based on landscape goals. The three phases to accomplish this are: 1) define the landscape goals, 2) define an objective function that measures attainment of the landscape goal and any constraining functions, and 3) develop a heuristic to maximize (or minimize) for the objective function.

Phase one of stage two – Define the landscape goals

Potential landscape goals include the same five stand goals but applied at the landscape scale (limiting fire hazard, limiting insect and wind-throw hazard, enhancing wildlife habitat, improving fish habitat, and providing a positive PNV). Other potential landscape goals are spatial in nature. Examples include ensuring X number of snags are left on every 40 acres or maximizing even-flow of timber in area Y while improving fish habitat in area Z. The original idea behind the Applegate Project was to give the Applegate Partnership an opportunity to define the landscape goals and for the project science team to develop some landscape goals as well. However, because of difficulties encountered in the model development there was not enough time to coordinate with the Applegate Partnership and establish a well-defined set of landscape goals they wish to model. This left the process of defining the landscape goals to the science team.

The science team chose two landscape goals:

1) Big Trees – Maximize the number of big trees (> 30" DBH) across a managed landscape.
2) ERA Constraint – Limit each 6th-field sub-watershed to an Equivalent Roaded Acre threshold (ERA – to be discussed in phase two).

Two scenarios were developed to encapsulate the above landscape goals (first scenario) and provide for an alternative landscape "goal" of leaving a landscape unmanaged (second scenario). These two scenarios serve as the modeling alternatives that will be referenced throughout this thesis (scenarios will always be capitalized in remaining text; landscape and stand goals will not):

1) **Grow Only** - No management activities allowed in any stand.
2) **Big Trees** – Incorporate the big trees and ERA constraint goals.

The landscape goals are applied to a spatial sub-set of the entire Applegate Watershed. I call this sub-set the "eligible" or "modeled" landscape. The modeled landscape is a result of identifying those cells for which the SafeD model will keep track of.

For the Grow Only scenario the Red Buttes Wilderness and all non-forested areas are the only areas not modeled.

The modeled landscape for the Big Trees scenario are those forested cells that:

1. Are in a LSR with an initial stand quadratic mean diameter breast height (QMDBH) <= 15"
2. Are on federal lands in identified stream buffers with an initial stand QMDBH <= 15"
3. Are not in the Red Buttes Wilderness

The Grow Only scenario was chosen for its use as a "baseline" to measure other scenarios against. The ideal of having a big tree goal on the landscape was positively received during several discussions with the Applegate Partnership. The ERA constraint goal was incorporated to allow me an opportunity to work with a multiple-goal scenario. ERA thresholds were also discussed with the Applegate Partnership although no formal guidelines or parameterization came from them.

Phase two of stage two – Define objective function and constraint functions

The Grow Only scenario does not require any objective function because no decisions need to be made; the landscape is left alone to grow. The big trees goal in the Big Trees scenario is formulated as a Model I nonlinear integer problem (see Literature Review) where decision choices are represented by binary integer (0-1) variables. In this case, the fundamental choice is whether or not to assign a particular stand prescription to a stand. The objective function is to maximize the number of big trees across the managed landscape. The objective function is formulated as:

maximize:

$$\sum_{t=1}^{n}\sum_{k=1}^{m}\sum_{j=1}^{q} r_{k,t,j}\, x_{k,j}$$

where:

t = period
n = total number of periods
k = stand (note: stands are 25 m. x 25 m.)
m = total number of eligible stands
j = prescription
q = total number of potential prescriptions
$r_{k,t,j}$ = total number of big trees (>= 30" DBH) in stand k in period t from prescription j
$X_{k,j}$ = 0-1 variable indicating prescription j is assigned to stand k

subject to:

INPUT CONSTRAINT

$$\sum_{j=1}^{q} x_{k,j} = 1 \quad \forall k$$

POLICY CONSTRAINT (sub-watershed disturbance)

A normalized numerical coefficient called the Equivalent Roaded Acre (ERA) will be used to track overall land disturbance within each 6th field sub-watershed (approximately 2,200 acres each). A road surface is considered to be the most extreme type of disturbance in terms of increasing or concentrating water flows and sediment production (Carlson and Christiansen, 1993). A road is given an ERA coefficient of 1.0. Other types of disturbance are equated to a road surface by ERA coefficients reflecting their relative level of disturbance. ERA coefficients are decayed over time to reflect recovery.

The ERA constraint function is formulated as:

$$\frac{\sum_{k=1}^{p} ERA_k}{p} \leq ERA\_Threshold_t \quad \forall t, w$$

where:

t	=	period
w	=	sub-watershed
k	=	stand in sub-watershed w
p	=	total stands in sub-watershed w
ERA_k	=	ERA value for stand k in period t
$ERA\_Threshold_t$	=	ERA threshold for period t

Phase three of stage two – Develop heuristic to maximize the objective function

A heuristic optimization algorithm was developed to select the prescriptions for each stand for the entire planning horizon to achieve the Big Trees landscape goal. Heuristic programming is a broad term used here to describe a method of solving large, multi-variable, combinatorial optimization problems. Simulated annealing, the great deluge algorithm, genetic algorithms, and tabu search are examples of heuristic algorithms (often just called heuristics). Heuristics find optimal or near-optimal solutions through various processes that allow the algorithm to employ a search or sampling strategy of the possible solutions. Some unique elements of heuristics include: 1) they do not look at all the possible solutions; 2) they may have some sort of "memory" which prevent the algorithm from continuously looking in one "area" for a solution; 3) they tend to have a set of rules that allows the algorithm to accept inferior solutions, to avoid being trapped in local optima, with the idea that it may lead to a better solution; and 4) they are not guaranteed to find the mathematically optimal solution, in fact they may not find a feasible solution at all (see discussion on heuristics in Literature Review).

I chose to use the Great Deluge Algorithm (GDA) as described by Dueck (1993). The GDA requires only one acceptance criteria parameter which makes it easier to implement than other similar techniques. As described in the literature review the GDA

rests on the notion that given a starting value, inferior solution values are acceptable subject to an increasing "level". The difference between the level and the best solution value ever found decreases as the algorithm progresses by a single parameter called "rain". This idea works for both minimization and maximization problems.

The solution given by the GDA heuristic will be a near-optimal solution to the underlying landscape question "Which prescription do I choose for each stand (from the available set of prescription alternatives developed by PREMO) that optimizes the landscape given a set of landscape goals." Figure 6-3 shows a flowchart for a generalized version of the GDA heuristic used.

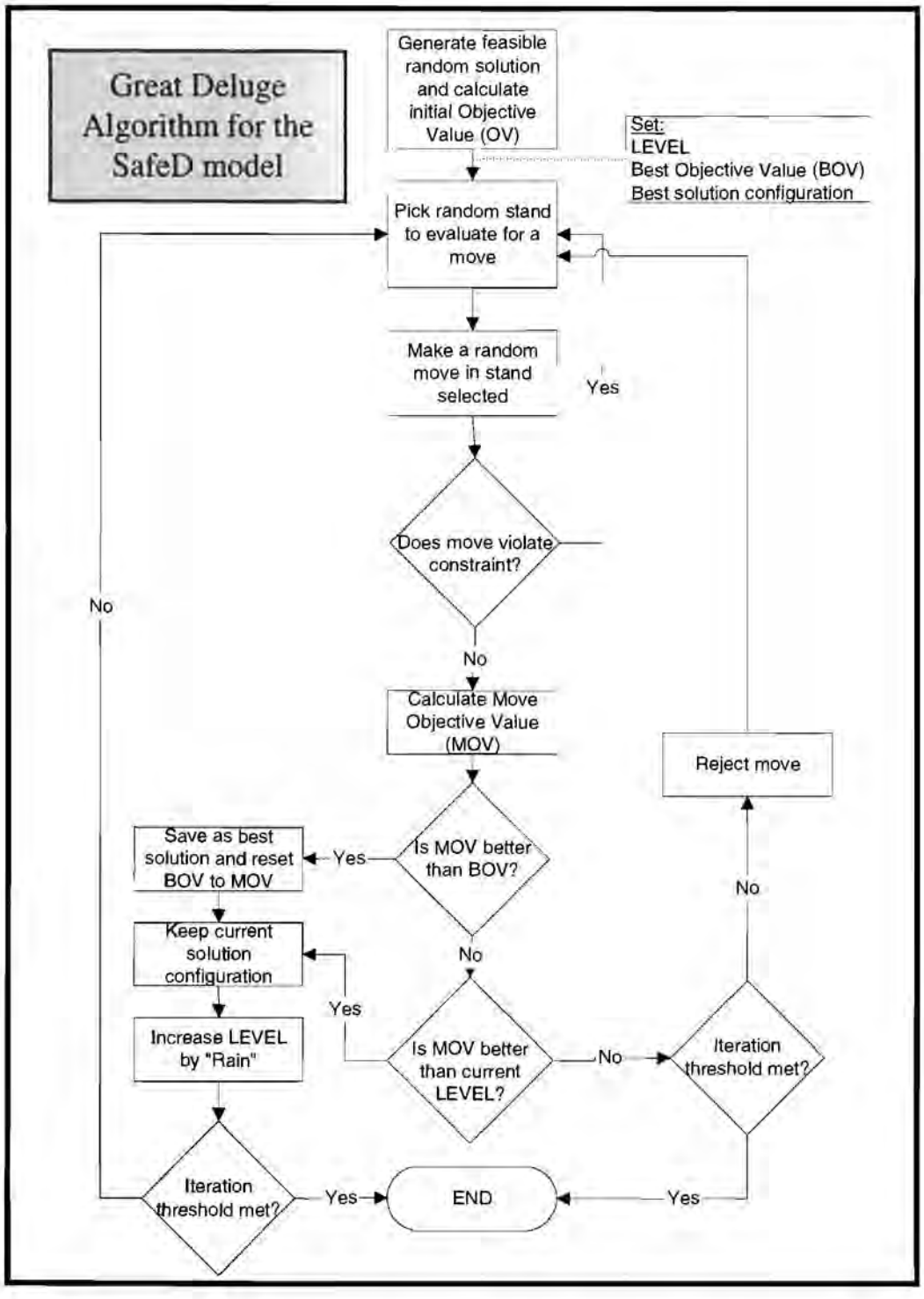

Figure 6-3: Flowchart of the Great Deluge Algorithm used by SafeD

Only the Big Trees scenario required a problem-solving heuristic. The Grow Only scenario had no decision variables; the landscape is left alone to grow without any management actions. The computer code written for the GDA heuristic is embedded within several functions of the overall SafeD model code. The entire code can be seen in Appendix F. A rough description of the GDA implemented for the Big Trees scenario follows:

> A completely random solution is generated. A solution is defined as the allocation of a stand prescription to each stand on the managed landscape. The Big Trees scenario had 2,289,823 stands that were eligible for management (353,491 acres). There were 19 different stand prescriptions that could be assigned to any single stand (see stage one). A random solution is generated by randomly assigning one of the 19 available prescriptions to each of the stands. Once that is completed the solution is evaluated to ensure it is feasible – in other words, it does not violate any constraint. The only constraint imposed for the Big Trees scenario was a sub-watershed ERA threshold. The SafeD model calculates an ERA value for each of the 218 sub-watersheds in the Applegate River Watershed. If any one of those sub-watersheds has an ERA value higher than the allowed threshold value the entire random solution is discarded and another generated. This process continues until a feasible random solution is obtained for the starting point.
>
> Once there is a valid initial solution the initial objective value (OV) is calculated. For the Big Trees scenario the OV is the total sum of all trees greater than or equal to 30" DBH on the managed landscape across the entire planning horizon. This value is initially calculated by looking at each stand, finding the appropriate treelist and summing the number of eligible trees. After the initial OV is calculated two key variables are set. First is the best objective value (BOV). Since the initial OV is the only one found it is also set as the BOV. Next set is the

LEVEL. The LEVEL is the allowable value below (for a maximization problem) which a new OV cannot drop. The initial OV, the BOV, and the LEVEL are now set and the GDA algorithm is ready to enter into a problem-solving iteration (looping) process.

The first step (or top) in the GDA loop is to make a "move". A move constitutes some change or deviation in the state of the solution. I chose to make neighborhood moves by randomly selecting one of the stands in the current solution and assigning a new random stand prescription allocation. The only restrictions in place for the move is that I can't pick the same stand twice in a row and I can't reassign the same prescription (with over 2 million stands to randomly chose from the first restriction was really unnecessary). The solution in place before making a move is always stored in case the move proves to violate any constraints. After the move is made the move solution is evaluated to see if does violate the ERA threshold constraint. If it does violate the ERA threshold constraint, the move is rejected, the stored pre-move solution is returned, and the next loop is executed. If the move does not violate the ERA threshold constraint the move objective value (MOV) is calculated using the objective function described earlier.

At this point the GDA has changed the solution from its previous state and the critical question is to decide whether or not to accept the move and continue the looping process or reject the move and continue the looping process. First the GDA asks, "Is the MOV better than the current BOV (better meaning a higher number) ?" If so, the current configuration of the solution is saved, the BOV is reset to the value of MOV, and the LEVEL is increased by the rain parameter. The rain parameter is a constant value determined through a trial-and-error process (see Dueck, 1993). A check is made to see if the GDA looping threshold has been reached. The looping threshold is another value determined through a trial-and-error process. If so, the GDA ends; if not, the GDA returns to the top of the loop and picks another random stand to evaluate for a move.

However, if the MOV is not better than the current BOV the GDA asks, "Is the MOV better than the current LEVEL value?". If so, the current configuration of the solution is saved and the LEVEL is increased by the rain parameter. Notice that the BOV is not reset this time. A check is made to see if the GDA looping threshold has been reached. If so, the GDA ends; if not, the GDA returns to the top of the loop. Lastly, if the MOV was not better than the current LEVEL value, the move is rejected, the stored pre-move solution is returned, and the GDA checks the looping threshold and continues as described above.

This marks the end of the three main phases of stage two. Once the GDA is completed and a solution is found the SafeD model needs to run through some internal C code to account for the solution variables found. This is a programming process and basically entails the storing of specific values for each stand within the computer's memory. Most of this data is related to information obtained from treelist which are stored on the computer's hard drive. Storing the information in memory allows the SafeD model to access the data faster and thus run faster in stages three and four.

Stage Three

Introduction

Stage three begins the movement from period to period for each period in the planning horizon. The Applegate Project was originally designed to be a 100-year planning horizon with twenty 5-year periods. Due to computer processing time limitations, this was later reduced to a 40-year planning horizon with eight 5-year periods. The programming implementation permits an unlimited number of 5-year periods – as long as certain stand and parameter data are available. Stage three is the core of the simulation part of the SafeD model. There are five main parts to stage three with many components to each part:

1. Determine weather
2. Adjust fuel loads
3. Apply insect disturbance
4. Apply fire disturbance
5. Hazard analysis

Before I begin the discussion on the five parts of stage three I need to clarify some details about how the SafeD model stores and handles data for the landscape. The concepts I describe next are important for understanding the processes that occur in stage three. The words cell and pixel are often used interchangeably – they refer to the same thing. A structure which can be visualized by a series of "cells" or "pixels" in row and column format is called either a grid- or raster-based structure, as seen in Figure 6-4.

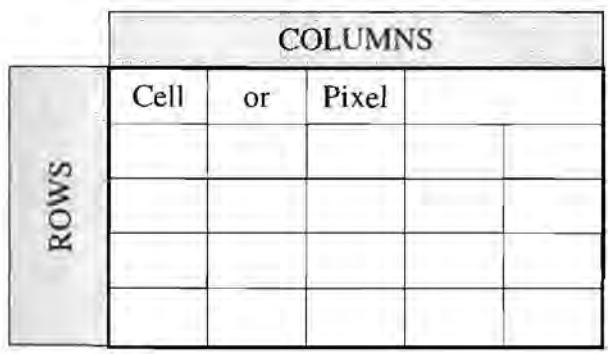

Figure 6-4: Example of a grid or raster structure

Each cell (or pixel) has the same dimensions (length and width). A landscape, such as the Applegate Watershed, can be broken up into cells by superimposing this grid. I have already discussed that the stands for the Applegate Project are 25 m. x 25 m. pixels. All of the landscape attributes stored within the SafeD model originates from data that is raster-based. I am using the term landscape attribute to refer to any attribute, either assigned or calculated, for each individual stand (or pixel in this case). Every cell for the Applegate Watershed has landscape attribute information stored in the SafeD model. Some of the landscape attributes do not change over the planning horizon (slope, aspect, elevation, PAG, etc.); and others change each simulation period (debris pools, fuel models, basal area, etc.). A complete list of the landscape attributes used within the SafeD model, a brief description of each, and the data source can be seen in Table 6-1.

Landscape attribute name in SafeD	Brief Description	Source*
Cellid	Unique value to identify each cell	GIS
Treelist	Lookup value to associate with treelist data	GIS
Elevation	Stored in meters	GIS
Aspect	0 – 360 degrees	GIS
Slope	Stored in percent	GIS
Goal	Lookup value to identify stand goal	SafeD generated
Owner	Code for ownership	GIS
PAG	Plant Association Group value	GIS
Allocation	Federal land allocation – n/a to private lands	GIS
Sub-watershed	Identification of 6th field sub-watersheds	GIS
Hold	Lookup value to identify stand goal	SafeD generated
Buffer	0-1 value to identify federal lands in stream buffers	GIS
Fire History	0-1 value to identify areas with know fire history	GIS
Initial Vegetation	Code indicating initial vegetation (1-13)	GIS
Initial Structural Stage	Code indicating initial structural stage (1-15)	GIS
Duff	Debris pool	PREMO & SafeD
Litter	Debris pool	PREMO & SafeD
Class25	Debris pool	PREMO & SafeD
Class1	Debris pool	PREMO & SafeD
Class3	Debris pool	PREMO & SafeD
Class6	Debris pool	PREMO & SafeD
Class12	Debris pool	PREMO & SafeD
ClassOver12	Debris pool	PREMO & SafeD
Fuel Model	Code to indicate fuel model	SafeD generated
ERA	Equivalent Roaded Acre value	SafeD generated
Flame Height	To store current periods flame height calculation	FARSITE
Basal	Stand basal area	PREMO
Closure	Stand canopy closure	PREMO
CBD	Stand crown bulk density	PREMO
HLC	Stand Height to Live Crown	PREMO
Stand Height	Stand Height	PREMO
Big Trees	Number of big trees (>= 30" DBH) in the stand	PREMO
Vegcode	Single code for both veg. class and structural stage	PREMO
CF Harvest	Cubic feet of harvest from the stand	PREMO

Source*:
GIS – data originated in a GIS system through various processes.
SafeD generated – data was generated internally within the SafeD model.
PREMO & SafeD – combination of data generated within PREMO and modified by the SafeD model.
FARSITE – third party program that generates fire information.
PREMO – data originated through the growth and yield process in PREMO.

Table 6-1: Landscape attributes used by the SafeD model

Determine weather

Weather is a stochastic variable within the SafeD model. The science team chose to use precipitation data obtained from a weather station in Medford, Oregon (just outside the Applegate Watershed to the east). The variation in precipitation was calculated from this data and three weather patterns were categorized: Wet, Moderate, and Drought. Within the drought-type pattern two levels exist: Mild and Severe. The weather pattern is currently used for two purposes: 1) the insects and, 2) fire disturbances (the weather role will be discussed in upcoming sections for these disturbances). Determining the weather pattern for any given period was based on a probability matrix and a random number generator.

The probability matrix for 5-year weather patterns in the Applegate Watershed is:

Wet	10%
Moderate	65%
Drought	25% (two levels of drought – see below)

A random number generator is used with the SafeD model. I will briefly describe the process here because random numbers are used a few more times within the SafeD model and I feel it is important to clarify how they work. The ideal behind probabilities and random numbers is to generate a number between 0 and 1, which can correspond to probability values of 0% to 100% (which if divided by 100 is 0 to 1). I ask the C language to generate a random number then compare that random number against the probability matrix to determine the weather pattern. If it was determined to be a drought period I would check what the weather pattern was during the previous period (which is stored in memory). If the previous period was also a drought period the current period was considered a severe drought; otherwise it was a mild drought. Only one severe drought period was allowed per 100-years. If a severe drought already occurred and the random number generator was calling for another one, I would cycle with new random numbers until either a wet or moderate weather pattern was found.

Adjust fuel loads

One of the experimental ideas the science team wanted to try in the Applegate Project was the calculation of fuel loads and the classification of each stand into fuel models based on current treelist data and fuel loads during each simulation period. We call this process "dynamic fuel modeling" to reflect the ideal that the fuel loads are changing each period because of stochastic disturbances, tree growth, tree and snag decay, and harvest which we cannot determine *a priori*. Science team members Agee (1999) and Bahro (personal comm.., various dates 1998-2000) developed the rules and processes for our dynamic fuel modeling; I was responsible for implementing their rules and processes into the SafeD model. To that end, I cannot discuss in detail the background or particulars in the development of dynamic fuel modeling. However, I will address what I implemented into the SafeD model. There are two components that I will be discussing in this section. The first is the accumulation and decay of debris pools into **fuel loads**. Second is the categorization of fuel loads into **fuel models**. It should be noted that much of the work done by the Applegate Project science team for dynamic fuel modeling was based on methodology from the Fire and Fuels Extension to the Forest Vegetation Simulator (FVS) by Beukema et al. (1998).

Most biomass from above-ground sources can be collected into debris pools as that material falls to the ground. Debris pools are diameter size categories for classification of this material. The SafeD model tracks the amount of material falling into each debris pool as well as account for decay of material already in the pools.

Initialization of debris pools for each stand really occurs once at the end of stage two – before the SafeD model enters stage three. Initialization is not related to the allocation of stand prescriptions; it is an independent process that happens regardless of what management actions are decided upon. The science team chose to initialize the debris pools based on the vegetation classification of each stand. Table 6-2 shows the initial values for each debris pool in each vegetation category in tons/acre.

	Debris pools (diameter size class in inches)							
Vegetation	Duff	Litter	0 - .25"	.25 – 1"	1 – 3"	3 – 6"	6 – 12"	> 12"
Barren	Not needed							
Water								
Shrub								
Grass/Forbs								
Red fir	30	.7	.7	2.6	3.6	5	4	5
Mixed conifer < 3000'	5	2.5	.2	.8	1.2	1	1	1
White fir	30	.6	.8	2.7	2.7	4.5	5	7
Pine	5	2.5	.2	.8	1.2	1	1	1
Closed cone pine	5	2.6	.3	.3	.4	1	2	3
Deciduous hardwood	2.3	1	.3	2.4	5	1	2	2
Conifer hardwood	5	2.5	.2	.8	1.2	1	1	1
Evergreen hardwood	3.7	1.8	.3	1.6	3.1	1	1.5	1.5
Mixed conifer > 3000'	10	1.4	.9	2.1	3.8	3	9.5	9.5

Table 6-2: Initial debris pool loadings (in tons/acre)

The science team then divided these debris pools into three fuel load categories: 1-hour, 10-hour, and 100-hour fuels. Not all the pools were used to categorize the fuel loads. Of the eight pools listed only four are needed to make the fuel loads:

- 1- hour = Litter + 0 - .25"
- 10-hour = .25 – 1"
- 100-hour = 1 – 3"

These three fuel load categories along with topographic and vegetation data are then used to assign a particular fuel model to each stand. A fuel model is a stylized and simplified description of fuel for a mathematical fire behavior model. The properties that

define fuel models include load and surface-area-to-volume ratio for each class (live and dead), fuel bed depth, and moisture of extinction (Anderson, 1982). Fuel model codes are standardized within the fire modeling community such that a fuel model 8 means something whether you are in the Pacific Northwest or Florida. Table 6-3 shows the fuel model (FM) classification matrix used for the SafeD model:

IF Vegetation is Barren, Water, Shrub, Grass/forbs
- Barren ..... FM 99
- Water ..... FM 98
- Grass/Forbs ..... FM 1
- Shrub
  - < 3000' ..... FM 4
  - > 3000' ..... FM 19

ELSE IF stand QMDBH < 5" or (stand QMDBH < 8.9" and canopy closure < 60%)
- > 3000' ..... FM 5
- < 3000'
  - vegetation is Pine ..... FM 2
  - vegetation is Deciduous hardwood ..... FM 17
  - others ..... FM 6

ELSE IF vegetation is deciduous hardwood that has < 60% canopy closure or a stand QMDBH of 5 – 8.9" and > 60% canopy closure ..... FM 6

ELSE IF 1-hour fuel load <= 1.5
- 10-hour < 1 ..... FM 18
- 10-hour 1 - 4.5 ..... FM 8
- 10-hour > 4.5 ..... FM 11

ELSE IF 1-hour fuel load 1.5 – 2.5
- 10-hour < 1 ..... FM 20
- 10-hour 1 – 1.9
  - 100-hour <= 1 ..... FM 2
  - others ..... FM 23
- 10-hour 2 –6 ..... FM 31
- 10-hour > 6 ..... FM 32

ELSE
- 10-hour < 1 ..... FM 9
- 10-hour 1 – 3
  - 100-hour <= 3.5 ..... FM 16
  - others ..... FM 10
- 10-hour > 3 ..... FM 12

Table 6-3: Fuel Model (FM) classification matrix

I have discussed two processes regarding dynamic fuel modeling: the initialization of debris pools and classification into fuel loads for the initial vegetation on the landscape (time 0 - start of the simulation – stage two); and the classification of fuel loads into fuel models regardless of the period. This brings me back to the dynamic fuel modeling processes that occur in stage three. There are basically three steps to the dynamic fuel modeling process that occur at the beginning of every period. Those three steps are: 1) decay debris pools, 2) add new contributions to debris pools, and 3) reclassify debris pools into fuel loads and those fuel loads into new fuel models.

Decay rates for the debris pools were established by Agee (1999) and implemented in the SafeD model as follows:

Duff and litter pools

- Every period (5-years), decay 2% of litter
- Decay 3% of duff for each year
- Take 16% of remaining litter and move it to duff

Remaining debris pools

Pool diameter size	Rate
0 -.25"	decay 12% per year
.25 – 1"	decay 12% per year
1 – 3"	decay 9% per year
3 – 6"	decay 1.5% per year
6 – 12"	decay 1.5% per year
> 12"	decay 1.5% per year

At the start of a period the SafeD model goes to each of the stands and decays the debris pools associated with the stand. This is done on a stand-by-stand basis. Once the decay step is completed, new net contributions to the debris pools are added. Net contributions to debris pools are a function of stand composition, structure, growth, mortality, and harvest. This information is actually calculated inside the PREMO program back in stage one as part of the prescription development. I will not address the calculations of net contributions in this thesis because if falls outside the scope of work I completed; I simply used the output data from PREMO (see Wedin, 1999). In essence, the SafeD model goes through the entire landscape on a stand-by-stand basis and

determines which PREMO output data is needed for the stand. The SafeD model stores the new net contributions for each stand in that period and now each stand has updated debris pool information – decay and new contributions are accounted for. The final step in the dynamic fuel modeling process is the recalculation of fuel loads based on the new debris pools and the further classification into fuel models as previously described.

During each simulation period there are two types of episodic disturbances that the SafeD model incorporates: insects and fire. The two components of stage three that I have just discussed (determine weather and adjust fuel loads) must occur every period before any type of episodic disturbance. Their function is to prepare and "update" the landscape attributes stored within the SafeD model such that when an episodic disturbance occurs the landscape attributes are current.

Apply insect disturbance

Another experimental idea the science team wished to try in the Applegate Project was that of having a stochastic insect disturbance regime which is spatial in nature. The rules for insect disturbance were based on expert advice and the intent was not to provide a precise set of triggers and impacts, but to simulate expected losses over the long run. Science team member Agee (1999) provided the research and rules for the insect disturbances.

The SafeD model embeds episodic mortality from insects in stochastic drought-related pulses. That is, only during drought periods is there episodic insect mortality. The rules for insect disturbances have two components: a basal area **threshold** is met and triggered; then a **severity** is applied. Three separate insect keys were developed:

	Key	Example of insects
1)	Douglas-fir	(Douglas-fir beetle, flatheaded borer)
2)	True fir	(fir engraver)
3)	Pines	(pine engraver, western and mountain pine beetle)

If a simulation period is assigned a wet or moderate weather pattern at the beginning of stage three then there is no insect mortality and the SafeD model skips the whole insect disturbance processes. Otherwise, both mild drought and severe drought weather patterns will trigger an insect disturbance. If a drought-type period is

encountered the SafeD model goes through the landscape on a stand-by-stand basis and determines if basal area thresholds are exceeded; if so, then severity (mortality) rules are applied. The basal area thresholds are a function of the stand PAG and which of the three insect keys is being evaluated (all three keys are evaluated – one at a time). Severity is applied to the individual treelist associated with the stand being evaluated. The insect disturbance rules for the three insect keys are shown in Appendix C.

Once the severity is applied to all the stands for each of the three insect keys the SafeD model runs through the landscape and re-calculates new stand level data for each of the stands hit by insects. Stand level data are single values for particular attributes which describe the stand as a whole. For example, a stand may have hundreds of individual trees, each having their own basal area; when pulled together as a stand those individual basal areas are collapsed into a single basal area value. PREMO calculates and outputs stand level data back in stage one (and the SafeD model reads and stores that data in stage one as well), but the SafeD model needs to make new calculations for each stand that encounters an episodic disturbance during the simulation periods. New stand level data that are calculated include: basal area, canopy cover, vegetation classification, structural stage, height, height to live crown, and crown bulk density. The particular methods used to calculate these values will not be addressed in this thesis. The origin of the equations used are embedded in the PREMO program (see Wedin, 1999).

Apply fire disturbance

The last episodic disturbance event that occurs every period is fire. Wildfire has played a significant role on the vegetative development in the watershed. The intent of the Applegate Project is to project past wildfire statistics into parameters that can be used for future wildfire simulations. Fire spread was accomplished using an external program called FARSITE (Finney, 1998). Science team members Agee and Bahro have provided the fire expertise for the project (this includes compiling past wildfire statistics and setting all the parameterization of data for input into FARSITE). The fire disturbance portion of the SafeD model is significant. My role in terms of this thesis is the preparation of data for input into Farsite and the execution of processes developed by the science team. I will detail that information generated specifically by the SafeD model

and I will attempt to make brief discussions on the rest. Figure 6-5 shows a general flow of the processes that occur within the SafeD model for the episodic fire disturbance:

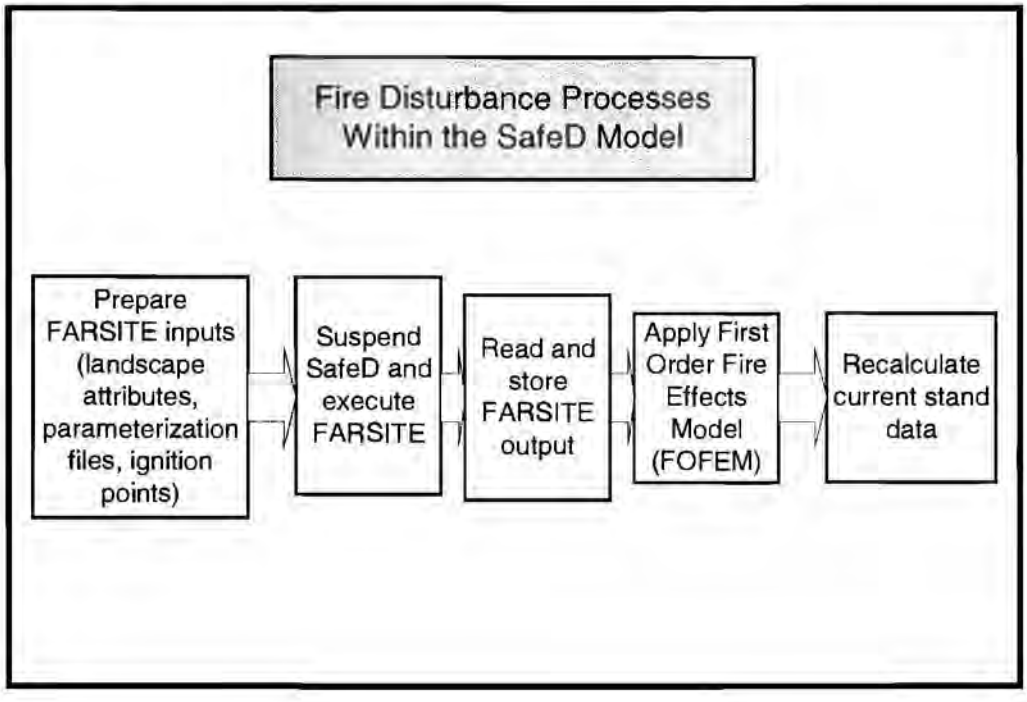

Figure 6-5: General flowchart of fire disturbance processes

Past Wildfire Statistics:

We have data on all wildfire activity since 1916 within the watershed boundaries. We also have data for wildfires on state and BLM lands that surround the watershed since 1960. Lastly, we have data on wildfires in the Rogue River and Klamath National Forest since 1960 and 1970 respectively (these two forests are in the proximity of the Applegate Watershed). Statistics assembled by Agee (1999) on all the historical data shows considerable variation in acres burned and number of fires, even when time periods and areas are standardized (Table 6-4) The highest variation is in acres burned per 5-year period, with lower variation in the number of fires per 5-year period.

	Variation in acres burned by 5-yr periods[1]	Variation in number of fires by 5-yr periods[1,2]	Percent of fires larger than 10 acres	Percent area burned >10 acres/year	Mean area burned per 5-yr period[3]
Applegate Watershed	178x	30x	--	--	2,158
Surrounding State and BLM	15x	2x	3.6	95	3,499
Rogue River N.F.	13x	2.4x	3.5	--	485
Klamath N.F.	67x	--	--	--	24,357

1 expressed as a ratio of the highest to lowest area burned in a 5-yr period
2 10-acre and larger fires only
3 expressed per 500,000 acres (approx. size of the Applegate Watershed)

Table 6-4: Historical fire statistics

Seven fire size classes were evaluated for further statistics. These seven classes are:

A = 0 – 0.25 acres
B = 0.25 – 10 acres
C = 10 –100 acres
D = 100 – 300 acres
E = 300 – 1000 acres
F = 1000 – 3000 acres
G = > 3000 acres

Using these fire size classes the science team looked at the size distribution and the area burned. Figures 6-6 and 6-7 show those results (note: not all the areas from Table 6-4 are included).

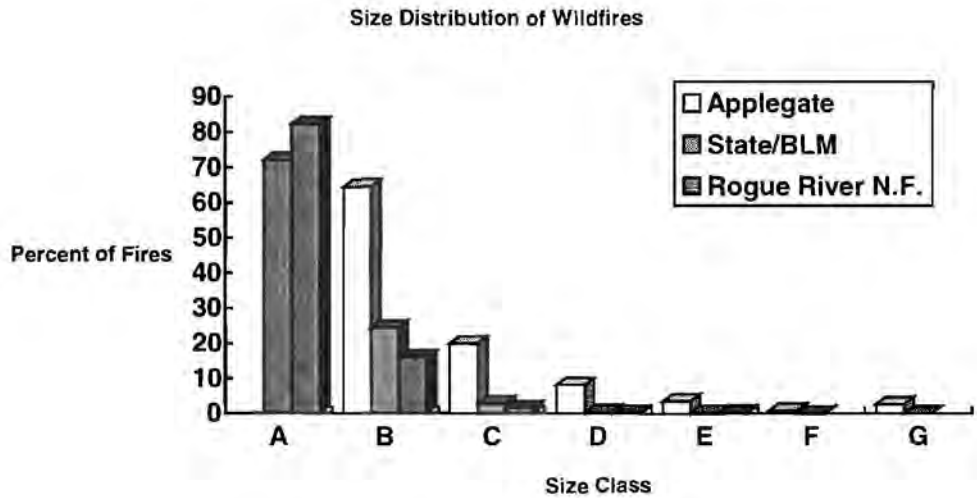

Figure 6-6: Size distribution of historical wildfires

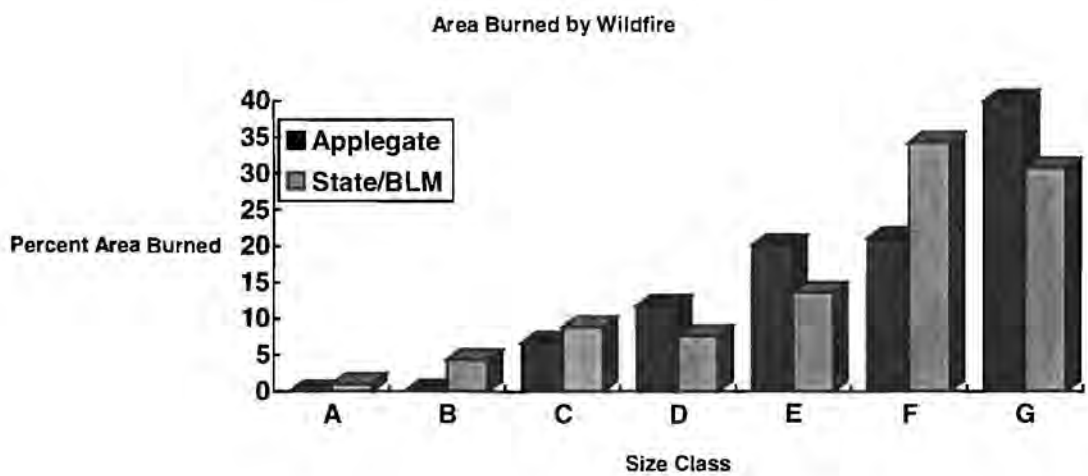

Figure 6-7: Area burned distribution of historical wildfires

The above information led to parameters which describe a "Base-Case Fire" scenario for each 5-year period. The Base-Case Fire is an attempt to parameterize past fire activity for describing the "best" scenario one could hope for in the future. In other

words, if simulations calculate fire statistics (over time) that mimic the Base-Case Fire scenario then this could be suggestive of a landscape that is encountering fire (and effects) similar to the historical data. The Base-Case Fire can then be used as a comparative against actual fire simulations within the SafeD model during the entire planning horizon. Parameters derived from the Base-Case Fire were used by science team members Agee and Bahro to calibrate input variables to the fire spread model FARSITE.

Introduction to FARSITE:

FARSITE (Fire Area Simulator) is a two-dimensional deterministic fire growth (or spread) model developed by Mark Finney, now of the U.S. Forest Service Fire Science Laboratory in Missoula, MT.. The Farsite model incorporates existing fire behavior models of surface fire spread, crown fire spread, spotting, point-source fire acceleration, and fuel moisture (Finney, 1998). A vector propagation technique for fire perimeter expansion that controls for both space and time resolution of fire growth is built into the FARSITE model. Vector fire perimeters (polygons) are produced at specified time intervals. The vertices of these polygons contain information on the fire's spread rate and intensity, which are interpolated to produce raster maps of fire behavior (Finney, 1998).

FARSITE Inputs:

The version of FARSITE used for this project is a DOS-based version that the SafeD model calls up. However, a number of spatial landscape attributes (in grid structure), an ignition location file, and other parameterization files must be created and ready for FARSITE to use each period. A list of significant input attributes and files is seen below in Table 6-5 (this is not an exhaustive list of FARSITE inputs):

	Type of Input	Responsible Source
Fuel Model	Landscape attribute	SafeD
Ignition Locations	Landscape metric (in vector format)	SafeD
Date and Time Info	Parameterization file	SafeD
Canopy Cover	Landscape attribute	PREMO
Crown Height	Landscape attribute	PREMO
Height to Live Crown	Landscape attribute	PREMO
Crown Bulk Density	Landscape attribute	PREMO
Elevation	Landscape attribute	Constant - GIS
Slope	Landscape attribute	Constant - GIS
Aspect	Landscape attribute	Constant - GIS
Weather Info	Parameterization file	Constant – science team
Wind Info	Parameterization file	Constant – science team

Table 6-5: List of major inputs to the FARSITE model

SafeD-Generated Inputs:

Only three items needed by FARSITE are generated specifically by the SafeD model each period (Fuel model, Ignition Locations, and the Date and Time File). The rest are either generated by PREMO or are constants developed through GIS or by the science team. I will first discuss the three inputs which the SafeD model is specifically responsible for.

Fuel Model

Fuel models provide the physical description of the surface fuel complex that is used to determine surface fire behavior. Inherent to fuel models are loadings (weight) by size class and dead or live categories, ratios of surface area to volume, and bulk depth (Finney, 1998).

Ignition Locations

FARSITE is used only once per 5-year period. The challenge the science team faced was to calibrate the past wildfire statistics into meaningful attributes that can be used to parameterize FARSITE inputs. In the case of ignition point locations a strategy was developed to allow the simulation of 5-year's worth of fires within a single run of

FARSITE. There are three components to this strategy: 1) determine how many fires there will be during the 5-year period, 2) determine how long they will burn, and 3) determine where the individual ignition points will be. The second component (how long they burn) will be discussed later when I detail information on the Date and Time File.

The initial idea to determine how many fire ignition points there are per 5-year period was to allow a range based on the weather pattern determined for that period (Agee, 1999). That ideal was later superceded by having a single range of ignition points regardless of the weather pattern (Bahro, personal comm., Oct. 1999). For my thesis work the range was determined as 5 – 15 ignition points per 5-year period. The actual number is selected by the SafeD model with a random number generator. It should be noted that ignition points are x,y coordinates (for each point) that FARSITE uses as the source of an individual fire. Within the SafeD model each cell (or stand or pixel) has an x,y value associated with the center-point of the cell. When I refer to a "point" in this discussion I am making an abstraction of a cell as a point represented by its center x,y coordinates.

Once the number of ignition points is determined, the next step is to allocate those ignition points on the landscape. A sequence of rules were developed to help guide the selection of each ignition point. For each of the *n* ignition points determined in a given period all of the below rules must be checked before the ignition point is accepted. A brief description of these rules is as follows:

1. A probability is assigned for the likelihood of an ignition point occurring in areas with no previous fire history.
2. Ignition points can not be located in water.
3. Probabilities are assigned to the likelihood of ignition points occurring within certain elevation bands as a function of the weather pattern.
4. Ignition points can not be located within one mile of landscape border.

<u>1) A probability is assigned for the likelihood of an ignition point occurring in areas with no previous fire history:</u>

Table 6-1 at the beginning of the Stage Three section shows an input landscape attribute called "fire history". The fire history attribute is a simple 0-1 variable (for each

cell) indicating whether or not we have data indicating the occurrence of previous fire activity within the cell. That information stems from the Past Wildfire Statistics section earlier. During the ignition point selecting process the fire history attribute is evaluated to determine if there has been previous fire activity in the cell (value 1) or not (value 0). If there has not been previous fire activity within the cell there is only a 60% chance that the cell is allowed to be selected. A 60% chance was determined through a trial-and-error process by the science team that produced a reasonable allocation of fire ignition locations within and outside of "fire history" polygons. Random numbers are used to compare against the given probability.

2) Ignition points can not be located in water:

A simple check is made to determine if the ignition point is being selected from water on the landscape. Several input landscape attribute themes have information regarding the presence or absence of water.

3) Probabilities are assigned to the likelihood of ignition points occurring within certain elevation bands as a function of the weather pattern:

For each of the three main weather patterns (wet, moderate, drought) a probability matrix is used to determine the likelihood of ignition point locations within certain elevation bands (Bahro, personal comm., June 1999). Table 6-6 shows this matrix:

Weather Pattern	Elevation Band	Probability
WET		
	>= 3,000 feet	10%
	1,500 – 3,000 feet	15%
	0 – 1,500 feet	75%
MODERATE		
	>= 3,000 feet	15%
	1,500 – 3,000 feet	35%
	0 – 1,500 feet	50%
DROUGHT		
	>= 3,000 feet	10%
	1,500 – 3,000 feet	45%
	0 – 1,500 feet	45%

Table 6-6: Ignition-Elevation probability matrix

The elevation at each potential ignition point is evaluated against this matrix to determine if the point is acceptable. Once again random numbers are used to compare against the given probability.

4) Ignition points can not be located within one mile of landscape border:

The final check for any potential ignition point is to ensure that the point is not located within one linear mile of the landscape edge. This is to give room for the fire to spread in any direction without immediately running off the landscape – where there is no data available. The one mile buffer is a value I determined from trial and error. The SafeD model allows for this distance to be modified.

Date and Time Information

The Date and Time file is a text file that sets some timing and resolution parameters for FARSITE to use. Included in this file is information on:

1. Date and time of fire start and fire end
2. Timestep
3. Distance resolution
4. Perimeter resolution

1) Date and time of fire start and fire end:

The science team chose to use the number of ignition points and the burning duration as the controllers in trying to mimic a Base-Case Fire. This is because these are two user-inputs to the FARSITE program. I have already discussed the strategy behind the number of ignition points. The burning duration is the second controller which specifies a starting date and time for the fires and a ending date and time. The date is important as another stochastic element (see Weather and Wind Files sections later).

Agee (1999) calibrated the initial rules for determining how long fires would burn as a function of the weather pattern assigned to the period. These rules were later modified by science team member Bahro (personal comm., April, 1999) as shown in Table 6-7.

Weather Pattern	Duration
Wet	24 hours
Moderate	48 hours
Drought-type	96 hours

Table 6-7: Fire duration times

2) Timestep:

FARSITE uses the timestep and spread rate of the fire to compute the distance traveled by fire at the vertices of the fire edge (recall that FARSITE uses a vector model of fire growth) (Finney, 1998). The timestep can be viewed as the maximum amount of time that landscape and environmental conditions are assumed constant so that fire growth can be projected. A timestep of 4 hours was found to satisfy our need for speedy computation without greatly affecting the precision or capabilities of FARSITE.

3) Distance resolution:

Distance resolution is the maximum horizontal spread distance allowed before new information from the landscape is required. It is the resolution in the radial spread direction. The distance resolution is also used by FARSITE to dynamically adjust the timestep to achieve a specified level of spatial detail. For example, assume a hypothetical

fire in which the timestep is four hours and the distance resolution is 200 meters. If after two hours the fire has burned in a radial distance of 201 meters then FARSITE will automatically reset the current timestep instead of waiting for the original four hours to complete. We used a distance resolution of 200 meters (eight 25 meter cells in a radial distance).

4) Perimeter resolution:

Perimeter resolution is the maximum distance allowed between vertices of the fire polygon (Finney, 1998). This value is related to the level of detail wanted to describe the outer perimeter of fires. A larger perimeter resolution results in coarser fire polygons. We used a perimeter resolution of 200 meters.

PREMO-Generated Inputs:

There are four landscape attributes needed by FARSITE which are generated by PREMO in stage one and stored for use during the simulation by SafeD: canopy cover, crown height, height to live crown, and crown bulk density (Table 6-8). Because the calculation of these attributes is done in PREMO I will not discuss how they are generated (see Wedin, 1999).

Landscape Attribute: Calculated in PREMO	Usage in FARSITE*
Canopy Cover	Used to determine an average shading of the surface fuels that affects fuel moisture calculations. It also helps determine the wind reduction factor that decreased wind speed from the reference velocity of the input stream (6.1 m above the vegetation) to a level that affects the surface fire.
Crown Height	Affects the relative positioning of a logarithmic wind profile that is extended above the terrain. Along with canopy cover, this influences the wind reduction factor, the starting position of embers lofted by torching trees, and the trajectory of embers descending through the wind profile.
Height to Live Crown	Used only with the surface fire intensity and foliar moisture content to determine the threshold for transition to crown fire.
Crown Bulk Density	Used to determine the threshold for achieving active crown fire.
* Source: This is a modification of Table 1 found in Finney, 1998.	

Table 6-8: PREMO-generated inputs to the FARSITE model

GIS-Generated Inputs:

There are three landscape attributes which I call constant (they don't change) and are generated through GIS processes before the simulation starts: elevation, slope, and aspect (Table 6-9). They are stored within the SafeD model during stage one and used whenever appropriate.

Landscape Attribute: Constant GIS-generated	Usage in FARSITE*
Elevation	Used for adiabatic adjustment of temperature and humidity from the reference elevation input with the weather stream.
Slope	Used for computing direct effects on fire spread, and along with Aspect, for determining the angle of incident solar radiation (along with latitude, date, and time of day) and transforming spread rates and directions from the surface to horizontal coordinates.
Aspect	See slope.
* Source: This is a modification of Table 1 found in Finney, 1998.	

Table 6-9: GIS-generated inputs to the FARSITE model

Science Team Generated Inputs:

There are five parameterization files that are needed by FARSITE and are created prior to the start of a simulation by the science team. These five files are also constant throughout the simulation although the possibility exists for them to be dynamically changed during the simulation. All five files are simple ASCII text files which can be created with any text/word processor.

Weather Information

The weather file consists of daily observations of minimum and maximum temperature and humidity, and of precipitation at a specified elevation (example in Table 6-10). Science team member Bahro produced the weather file we used from weather data collected within and around the Applegate Watershed. The dates were confined to a range from August 15th to September 13th. In theory FARSITE will accept a weather (and wind file) with information for every day of the year. The science team felt that it was important to restrict the weather and wind dates to the August-September timeframe to help attain fires that mimic the Base Case Fire scenario (Agee, 1999). The SafeD model does have a stochastic element built into the date selecting process. A random number generator selects (for a single run of FARSITE during a given simulation period) the actual starting date (confined to the above timeframe). Additionally, three weather files and three wind files were created; one for each of the three main weather patterns (wet, moderate, drought). The weather and wind files used depends on the weather pattern assigned to the simulation period.

The weather file data is used to generalize a diurnal weather pattern for a designated portion of the landscape so that dead woody fuel moistures can be calculated (Finney, 1998). Adiabatic adjustment from the input elevations to any cell on the landscape determines the local temperature and humidity (Finney, 1998).

Month	Day	PPT	Hour AM	Hour PM	Temperature Min	Temperature Max	Relative Humidity Max	Relative Humidity Min	Elevation
8	15	0	500	1300	51	65	88	28	3500
8	16	0	400	1300	53	67	91	26	3500
8	17	0	500	1300	56	57	82	25	3500

Table 6-10: Example of information found in a weather file

Wind Information

The wind file consists of observations of wind speed, wind direction, and cloud cover (example in Table 6-11). As with the weather file, the wind file data is assumed to apply uniformly to the landscape. Wind inputs are required to reflect "open" conditions at 6.1 meters above the top of the vegetation (Finney, 1998). Wind speed is assumed to be parallel to the terrain and for forested terrain the open wind speed is reduced locally by the canopy cover landscape metric (Finney, 1998). Science team member Bahro developed the wind file used in the simulation.

Month	Day	Hour-Minute	Open Wind Speed *(mph)*	Wind Direction *(degrees Az)*	Cloud Cover *(percent)*
8	15	200	2	159	0
8	15	2200	7	29	0
8	16	400	7	162	0

Table 6-11: Example of information found in a wind file

FARSITE Execution:

I have discussed the fire spread model FARSITE and all the significant inputs to run the model. FARSITE is an external DOS-based program that must be called up by the SafeD model to run. That is accomplished by sending an "execution" statement to the computer which temporarily suspends the SafeD model and starts the FARSITE model. However, before that is initiated all the necessary landscape metrics, parameterization files, and the ignition point locations must be created and in the format required by FARSITE.

FARSITE has the capability to output a number of landscape raster files that describe fire spread parameters. These output files include:

1. Time of arrival
2. Fireline intensity
3. Flamelength
4. Rate of spread
5. Heat per unit area
6. Crown – No crown
7. Spread direction

Of these seven outputs only the flamelength output is utilized by the SafeD model. The flamelength is used to apply specific mortality rates to the individual treelist associated with stands affected by fire. The other six outputs describe other characteristics associated with a fire and they are being considered for inclusion in future work of the SafeD model (they may be helpful in the parameterization of future inputs).

First Order Fire Effects Model:

The First Order Fire Effects Model (FOFEM) is a model that quantifies the direct or immediate consequences of fire (Reinhardt et al., 1997). These consequences include: tree mortality, fuel consumption, mineral soil exposure, and smoke. The science team chose to use the FOFEM to quantify tree mortality as a result of flamelength.

FOFEM mortality tables were developed that describe a mortality rate for specific species with specific diameters given a specific flamelength. Those tables can be seen in Appendix D. This information is related to the treelist associated with any stand at the time a fire occurs on it. The process works like this:

1. A fire occurs during a simulation period. FARSITE outputs a flamelength grid specifying the actual flamelength associated with those cells in which fire occurred.
2. That information is read into the SafeD model which in turn identifies which treelist is associated with each cell affected by fire. Those treelists are "gathered" up to have FOFEM applied to them. A treelist will typically have multiple

species of trees associated with it. Each species has different fire mortality rates associated with it

3. Once #2 above is determined the SafeD model goes through each individual record in the treelist and applies a specific mortality rate as a function of the species, diameter, and flamelength (as seen in Appendix D).
4. The mortality records calculated are appended to the end of the treelist and given a code to indicate the record is now a snag.

Recalculate Current Stand Data:

Once the FOFEM effects are applied to all the stands affected by fire during the simulation period the SafeD model runs through the landscape and re-calculates new stand level data for each of those stands. This is the same process as described at the end of the Apply Insect Disturbance section.

This ends the processes that occur in stage three of the SafeD modeling strategy. The science team developed rules to include a wind-throw episodic disturbance but I have not included that into the SafeD model at this time. Wind-throw generally occurs in the higher elevations and the mortality is relatively insignificant compared to insects and fire. Future work on the Safe model could include wind-throw disturbance.

Hazard Analysis

The SafeD model incorporates two hazard analyses - insect hazard and flame hazard. However, at this time the hazard analyses are not used for any processes or evaluations that occur within the SafeD model. They were an experimental idea for the project to which significant time was devoted to developing code to accomplish them. The analyses are done at specific times during stage three and the outputs are used for mapping purposes only. Appendix E details the hazard analysis processes. Future work on the SafeD model should include looking at the hazard analysis during each period and adjusting activities to mitigate areas that are at high risk.

Stage Four

Stage four is the "adjustment stage". The flowchart seen back in Figure 6-1 lists the three main components of stage four as:

1. Use PREMO to develop new prescriptions for all stands affected by disturbances in stage three.
2. Adjust activities on the landscape to account for effects of disturbances in stage three.
3. Return to stage three for the next period until end of planning horizon.

The original idea behind stage four was to track all the stands that were affected by episodic disturbances during the period (either insect and/or fire) and track which 6th-field sub-watershed those stands were in. Then PREMO would generate a new array of prescriptions for each effected. Landscape goals, at the sub-watershed level, would be developed (or given) for which a new heuristic would be used to chose the allocation of prescriptions to the effected stands. However, time did not allow me to implement this strategy into the SafeD model.

Stage four has been simplified to the following steps:

1. Determine which stands were hit by either insects, fire, or both.
2. Let PREMO generate a new prescription for each stand that reflects the current prescription already assigned to the stand (i.e., it has the same stand goal and timing-choice assignment as discussed in the Stage One section).
3. Store the new prescription information for each stand and return to the beginning of stage three for the next period.

The modified strategy for stage four is a simple "prescription in – prescription out". Any stand on the landscape hit by an episodic disturbance during the period is tracked. At the end of the period all the stands hit are evaluated to find the number of new unique stand prescriptions that need to be re-generated. The variables that describe a new unique stand prescription are: the current stand prescription; which insects affected the stand, if any; and the flame length associated with the fire that hit the stand, if any. All three of

these variables must be identical for separate stands to receive an identical new prescription. PREMO is then used to generate a single prescription for each unique stand combination that has the same stand prescription allocation and timing-choice assigned prior to the disturbance.

Once the period representing the end of the planning horizon has been reached the simulation component of the SafeD model is complete. The SafeD model will calculate landscape variables such as acres per vegetation and structural stage class, how many acres were effected by disturbances in each period, the level of harvest, and a few others that will be discussed in the Sample Application section.

The SafeD Software Program

The SafeD model was written in the C and C++ language. Appendix F is a print-out of the full code used for this thesis. The over 12,000 lines of code are written in a hierarchical structure that starts from a function called *Main*. The *Main* function is the controller function which calls up other controlling functions that are specific to the tasks needed. I started the coding process in November 1998 and finished in January 2000. The code is a DOS-Windows based program with no user interface at this time. Any modifications or changes to variables must be accomplished within the un-compiled code and then recompiled before executing the program.

SAMPLE APPLICATION

The results I will discuss are an example of an application of the SafeD model for the Applegate River Watershed. The purpose of this example is two-fold:

1. To exercise the SafeD model to see how it works.
2. To begin to understand the relationships modeled for the Applegate River Watershed.

Understanding these two purposes is very important to understanding the results found in this section. In the Literature Review I discussed the limitations of other landscape models to address the goals set for the Applegate Project. This led to the idea of developing a hybrid optimization/simulation model (i.e. the SafeD model). But the processes and steps to build such a model were not in place – the science team needed to develop them as the model itself was developed. In the end there was an amount of uncertainty as to whether or not the SafeD model would function as designed. In other words, could the model do what it was designed to do? This sample application will demonstrate how the SafeD model functions.

The second purpose of this example is to begin to understand the relationships modeled. Those relationships, for example, may pertain to the growth and yield functions, the episodic disturbances, or the optimization process. To understand these relationships there must be an application such as the example I will discuss in this section.

I will use the term "simulation run" to express the notion of executing the SafeD model from start to finish. I will also frequently use the word "cell" to reference a stand in the discussion on landscape optimization.

One of the Desirable Model Characteristics shown in Table 2-1 is the ability for a model to have repeated simulations to assess variability. Repeated simulations are appropriate if there are some stochastic or probabilistic elements to the model. The episodic disturbances in the SafeD model are driven by stochastic weather patterns. Also, the FARSITE fire spread model has a stochastic element that determines the torching behavior of fires. The SafeD model has the capability to produce "repeat simulations" whereby non-stochastic elements can be re-used by the model for *n*

simulations – and allow the stochastic elements to change. The resulting data can then be used to assess variability. The results I will discuss in this thesis are the product of a single "typical" run of the SafeD model. I actually made five simulation runs of both the Grow Only and Big Trees scenario. However, this was in part a testing process. I needed to ensure that the model itself worked as expected. The science team as a whole is responsible for assessing the variability of the outputs and, as I have discussed, the science team has not had that opportunity.

Prescription Generation (stage one)

The initial landscape on the Applegate Watershed is broken into 133 unique stand types based on vegetation classification, structural stage classification, and area type (see discussion in Stage One of The SafeD Model). These 133 unique stand types exist regardless if the model is run for the Grow Only or Big Trees landscape goal. It takes PREMO 37 minutes to generate 19 prescriptions for each of the 133 unique stands. A total of 2,527 prescription files are generated and stored in this time.

Initial runs of PREMO (using a variety of prescription choices) showed stand basal areas that were much higher than expected. I tried to determine why that was happening but was unsuccessful. It is likely that there was not enough periodic mortality occurring within PREMO. Yet this was consistent with a strategy the science team was trying to accomplish in testing whether or not stand mortality could be better represented by episodic disturbance events. Without the episodic mortality occurring (in the SafeD model) the periodic mortality (in PREMO) is expected to be somewhat low. However, the cause-and-effect links have not all been determined and I was skeptical of leaving such high stand basal areas. To compensate I increased the periodic mortality rate inside PREMO.

The mortality increase I specified in PREMO greatly affected the number of big trees. The mortality functions I used selects (i.e. kills) trees in a descending order of tree diameter – starting with the tallest tree in a treelist. 67% of the mortality from the PREMO function I used comes from trees greater than 20" DBH and the remaining mortality occurs in trees less than 20" but greater than 8" DBH. This did have some

desirable results (stand basal areas were reduced to a more expected level) but the effect on the number of big trees was greater than desired.

Landscape Optimization (stage two)

The Modeled Landscape

I described earlier the idea of the "modeled" or "eligible" landscape in regards to the two scenarios demonstrated in this thesis. This modeled landscape is really a function of how I chose to store, track, and count information regarding the scenario being modeled. For the Grow Only scenario the Red Buttes Wilderness and all non-forested areas are the only areas not modeled. There were a total of 2,637,289 cells that were modeled (407,131 acres). The modeled landscape for the Big Trees scenario was developed using different rules (see discussion in Stage Two of The SafeD Model). The result was 2,289,823 eligible cells for 353,491 acres. For the analysis of the Big Trees scenario I could have chosen to track information for those acres that were included in the Grow Only scenario but not the Big Trees scenario (407,131 – 353,491 = 53,640 acres). This capability was not programmed into the SafeD model but should be included in future work.

The Great Deluge Algorithm (GDA)

The GDA was used only for the Big Trees scenario. The problem was solved ten times during the development process. The rain amount, the looping threshold, and other move strategies were tried during this process with similar results. I used a single run of the GDA as the basis for the remaining simulation work. Note that the remaining discussion on the GDA is based on this single run (again, which was similar to the previous 10 developmental runs).

The GDA found a solution for the Big Trees scenario in 8 hours. A rain amount of 0.001 was found to be effective during the GDA development. A looping threshold of 29,767,699 feasible moves was set through a trial-and-error process. Feasible means that the move does not violate any constraint; it may or may not be a better move. This works

out to a potential for each eligible cell to move 13 times. In terms of computer processing time there are an average of 1,033 moves evaluated every second.

The solution found by the GDA for the Big Trees scenario is spatially and temporally feasible with regards to the big trees goal and ERA constraint. However, I cannot guarantee any level of optimality for the solution found. Several factors stand in the way of obtaining and/or measuring the success of the GDA:

1. A lack of independent solutions.
2. Recognition that the GDA parameters need more calibration.
3. Uncertainty that Objective Function is sufficient for the problem size.
4. Uncertainty with input stand data.

A lack of independent solutions:

I ran and documented only one solution of the GDA for this thesis. Statistical inference methods which can be used to validate heuristic solutions require multiple solutions. Because the landscape optimization portion of the SafeD model is only one component of the overall SafeD model I chose not to spend additional time in obtaining multiple solutions. Future work on the SafeD model should include processes to obtain multiple solutions and use statistical inference to describe the optimality of the solution.

Recognition that the GDA parameters need more calibration:

The rain amount, looping threshold, and move strategy I used for the GDA were satisfactory but not necessarily the best. At the end of the 8-hour run the GDA was still occasionally finding improving solutions. I noticed the same phenomenon during the developmental runs. This could indicate that I need to increase the looping threshold (which determines when the GDA will stop). I did try varying the rain amount and varying the move strategies during the developmental runs but saw no significant improvement. The cause-and-effect linkages between all the parameters are difficult to interpret so I would anticipate that the searching strategies could be improved.

Uncertainty that Objective Function is sufficient for the problem size:

The formulation of the Objective Function may not be sensitive enough to measure significant changes in the Objective Value (OV) as the result of a move on the landscape. The OV is a straight calculation of the number of big trees across the managed landscape during all time periods. A move constitutes changing the prescription allocation of one stand. To evaluate a move the Objective Function subtracts out the number of big trees associated with the pre-move prescription for the stand and adds in the new big trees for the new prescription. Because the stands are so small (25 m. x 25 m. = 0.15 acres) the number of big trees in each stand is very small as well. However, the number of big trees across the landscape is very large. It is possible that the GDA is continuously accepting inferior moves because making any move changes the OV so minutely (either positively or negatively) that it is always above the GDA threshold LEVEL for acceptance. This could also be a precision problem in that the number of decimal places used in the Objective Function is insufficient. There is another way to illustrate this problem. There are 2,289,823 stands in the solution with each having 19 potential prescriptions allocations. I am taking one stand and changing it to one new prescription. I see this with an analogy of dropping one drop of oil in a swimming pool and asking "can you detect the oil?".

Uncertainty with input stand data:

For the Big Trees scenario the GDA required data from PREMO regarding the number of big trees associated with all the prescriptions. I have already discussed that the big trees data from PREMO is questionable. The way in which the data is biased will influence the GDA results. If all the prescriptions are equally biased then perhaps the relative proportion of big trees with the current data is good enough to use. It is more likely that some prescriptions will have a far greater number of big trees and some prescriptions will have far less than the prescriptions I used for this simulation. Without further investigation into the PREMO calibration, conclusions should be seen as tentative.

ERA Thresholds

There were no ERA thresholds placed on the Grow Only scenario. As mentioned in the ERA discussion (in Stage Two of The SafeD Model), ERA values account for vegetation alteration that could affect soil erosion and/or peak flows caused by management activities and fire – and there are no management activities in the Grow Only scenario.

For the Big Trees scenario various ERA thresholds were tried. During the development process I tried two strategies: 1) keep a constant ERA threshold for all sub-watersheds over all time periods, and 2) have a ERA threshold vary with each period but applied equally to all sub-watersheds. There was no guideline predicating which strategy I should use so for the final simulation run I chose the latter strategy. My decision was based on the fact that I felt this strategy better represented the idea of allowing more activity to occur in the earlier periods than in the latter periods. Given the current conditions of the watershed (overstocked, high densities), heavier activities may need to occur in the earlier periods to reduce current fire hazards and to achieve the big trees stand goal.

The ERA threshold values I used were based on trial-and-error. I used ERA values from the SNEP project (SNEP, 1996) and Carlson and Christiansen (1993) to establish a rough starting point. During the development of the GDA I would lower the ERA threshold values to find the lowest threshold values (per period) that allowed the GDA to find solutions. Once the GDA started having a difficult time finding solutions with a set of ERA thresholds I stopped lowering the values (Table 7-1). The solution found does not violate the ERA threshold in any period for any sub-watershed

	ERA Threshold (for each sub-watershed)
Period 1	0.12
Period 2	0.12
Period 3	0.12
Period 4	0.12
Period 5	0.12
Period 6	0.10
Period 7	0.09
Period 8	0.09

Table 7-1: ERA thresholds used for the Big Trees scenario

Prescription Allocation

This section describes the allocation of stand prescription choices during stage two of the simulation process. The Grow Only scenario had no management choices. The Big Trees scenario tried to allocate prescriptions across the landscape to maximize the number of big trees (>= 30" DBH) over the entire planning horizon on those stands eligible for management. Recall from the discussion in Stage One that there are 19 prescription choices for each stand:

1. Reduce fire risk
2. Reduce insect risk
3. Enhance fish habitat
4. Enhance wildlife habitat – complex structure
5. Enhance wildlife habitat – simple structure
6. Maximize PNV
7. Reduce fire and insect risk, and maximize PNV
8. Enhance fish and wildlife (complex) habitat, and maximize PNV
9. All goals with emphasis on maximize PNV
10. Grow only

11 – 19. Same as first nine except harvest not allowed until 3rd period

The GDA has the ability to implicitly consider a large number of prescription allocation combinations across the managed landscape; limited by the looping threshold. The Big Trees scenario demonstrated in this thesis consists of over 2.2 million stands (353,491 acres) with 19 prescription choices for each stand. The number of unique prescription combinations that are possible to allocate can be expressed as: X^Y

where:

X = the number of prescription choices (19)
Y = the number of unique stands (2,289,823)

The GDA allocated the prescriptions as seen in Figure 7-1 (prescriptions 11-19 are grouped with its corresponding 1-9 value):

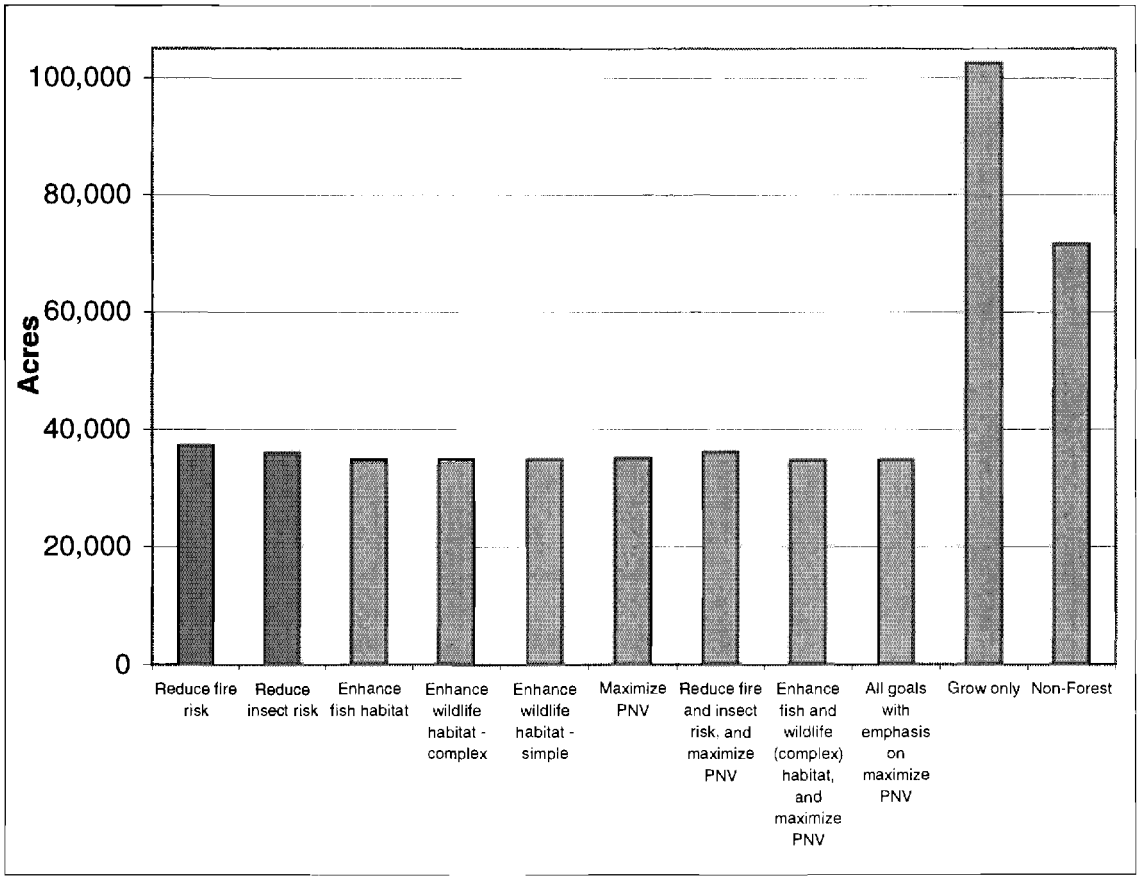

Figure 7-1: Prescription allocation for the Big Trees scenario

There was an almost even allocation of prescriptions 1 through 9 across the landscape. The grow only prescription allocation is larger because it includes acreage for those forested areas that were not in the eligible landscape (e.g. the Red Buttes Wilderness). The non-forest acreage (71,552) remains constant for either scenario.

There are two possibilities which may explain the results demonstrated. First, the method which I used to track and graph the prescriptions combined two equivalent prescriptions which differed only in their timing choice. For example, the "reduce fire risk" prescription shown in Figure 7-1 is actually a combination of the acres associated with a reduce fire risk - timing choice 0 (harvest allowed starting immediately) with the reduce fire risk – timing choice 3 (harvest allowed starting in 3 periods). It may be that by breaking and tracking the prescriptions and their timing choice separately (for each of the stand goals) that a more informative reflection of the allocation can be seen.

Secondly, the apparent even allocation of prescriptions may be a product of the randomization processes that occurred within the GDA. An initial feasible random solution is generated by the computer at the start of the GDA. There are no constraints on the randomness. The computer language used to code the SafeD model (C and C++) bases its random number generator on a even-distribution. Given that, one would expect every stand prescription to be equally represented in the initial random solution. The objective of the GDA was to search the "neighborhood" of this initial solution. The GDA accomplished this searching by randomly changing the prescription allocation of a single stand (called a "move"). The GDA process I used made only 29,767,699 moves. On average this allowed each stand in the solution to be moved 13 times. I cannot assure that happened. It may be that a great number of stands were never evaluated for a move. Of the 29 million moves that were made the new prescriptions evaluated were still based on a even-distribution random number generator.

The two possibilities just discussed and the uncertainties associated with the GDA make interpretation of the prescription allocation difficult. Visual examination of the prescription allocation revealed no obvious spatial correlation.

Number of Big Trees

The number of big trees per period was calculated for both scenarios before and after the actual simulation part of the model. The before-simulation calculation represents the big trees that are anticipated on the landscape given no episodic disturbances (Table 7-2). This value was calculated in the Objective Function and is the optimized value from the optimization part of the SafeD model in stage two. The after-simulation calculation represents the actual big trees found on the landscape after all episodic disturbances (in stage three) are accounted for (Table 7-3).

	Grow Only Scenario		Big Trees Scenario	
	Total	Per-Acre	Total	Per-Acre
Period 1	1,244,286	3.06	1,078,513	3.05
Period 2	726,171	1.78	871,672	2.47
Period 3	415,453	1.02	768,810	2.17
Period 4	252,352	0.62	511,887	1.45
Period 5	176,376	0.43	317,953	0.90
Period 6	123,648	0.30	284,984	0.81
Period 7	95,577	0.23	253,853	0.72
Period 8	84,382	0.21	244,375	0.69

Table 7-2: Number of big trees before-simulation

	Grow Only Scenario		Big Trees Scenario	
	Total	Per-Acre	Total	Per-Acre
Period 1	1,244,286	3.06	1,078,513	3.05
Period 2	726,612	1.78	871,587	2.47
Period 3	416,374	1.02	768,885	2.18
Period 4	235,916	0.58	498,023	1.41
Period 5	156,999	0.39	269,517	0.76
Period 6	110,320	0.27	273,074	0.77
Period 7	85,964	0.21	221,834	0.63
Period 8	76,564	0.19	183,562	0.52

Table 7-3: Number of big trees after-simulation

The Grow Only scenario did not actively manage for big trees but the information is useful as a comparison. The values shown are the total number of big trees, per period, within the modeled landscape only. The modeled landscape acreages are different for the two scenarios (see earlier discussion in The Modeled Landscape section) so the values are normalized to a per-acre value. Values for each period are calculated after growth for the period but before the occurrence of any episodic disturbance. Therefore the values in period 1 do not change for the before and after-simulation calculations.

In period 1 the number of big trees per-acre is nearly identical for the two scenarios. This would make sense because at period 1 there has been only 5-years of growth and the two scenarios are fairly similar in their modeled landscape. For the remaining periods the Big Trees scenario has a greater number of big trees per-acre than the Grow Only scenario. The main difference is the Big Trees scenario allocated prescriptions across the landscape that include timber harvesting and the Grow Only scenario did not. Without greater evaluation of the cause-and-effects of all the processes within the SafeD model I think a simple correlation can be made in regard to timber harvesting and the number of big trees: timber harvesting (as implemented within the PREMO prescriptions) increases the number of big trees because the growing space is allocated to fewer trees. PREMO does have restrictions on the level of harvest and rules regarding the silvicultural methods used (e.g., whether the harvest comes from "below" or "above"; and which species) (see Wedin, 1999).

The notion that timber harvest increases the number of big trees is not unusual. Assuming that a harvest is not a clear-cut (clear-cuts were not allowed by PREMO) then the stand will experience an "opening-up" effect. In general this allows for greater diameter growth of the residual stand. This is in contrast to an young, unmanaged stand (i.e. grow only) where the stand components may be forced to compete in a vertical nature because there isn't enough room to grow in a horizontal nature. This tends to lead to less diameter growth.

The big trees results illustrate the PREMO mis-calibration that I discussed earlier. The number of big trees drops from over 1.2 million to just over 84,000 in the before-simulation, Grow Only scenario. This is not expected. I would expect that given an un-managed landscape the number of big trees would increase. However, as I already

mentioned, I believe I adjusted the PREMO mortality function to such a degree that too many big trees died. An examination of all the big trees results show a substantial decrease in the number of big trees over the 40-year planning horizon. However, relatively speaking, the Big Trees scenario did seem to produce a greater number of big trees per-acre in both the before- and after-simulation analysis.

Harvest Levels

The Big Trees scenario does allow timber harvest to occur. Timber harvesting is not the goal of the Big Trees scenario – rather timber harvesting occurs as a by-product of the landscape optimization which is driven by the landscape goal to produce big trees across the landscape. The Grow Only scenario precludes timber harvest on the modeled landscape. This scenario was developed to serve as a baseline in comparing what happens to a landscape in which management activities are allowed to take place versus a landscape that is left unmanaged.

All of the prescriptions developed by PREMO in stage one use timber harvesting as an activity to achieve their respective stand goals (except for the grow only stand prescription) (Wedin, 1999). Therefore, for the Big Trees scenario, all of the prescriptions selected (described above in the Prescription Allocation section) have harvest levels associated with them. Those levels are tracked by the SafeD model (Figures 7-2 and 7-3). As with the big trees results there are two occasions when I evaluate the harvest levels: 1) before-simulation, and 2) after-simulation. The before-simulation values represent the predicted level of harvest given no episodic disturbances. The after-simulation values are the levels after episodic disturbances occurred on the landscape and after management activities were adjusted to account for the effects of those disturbances.

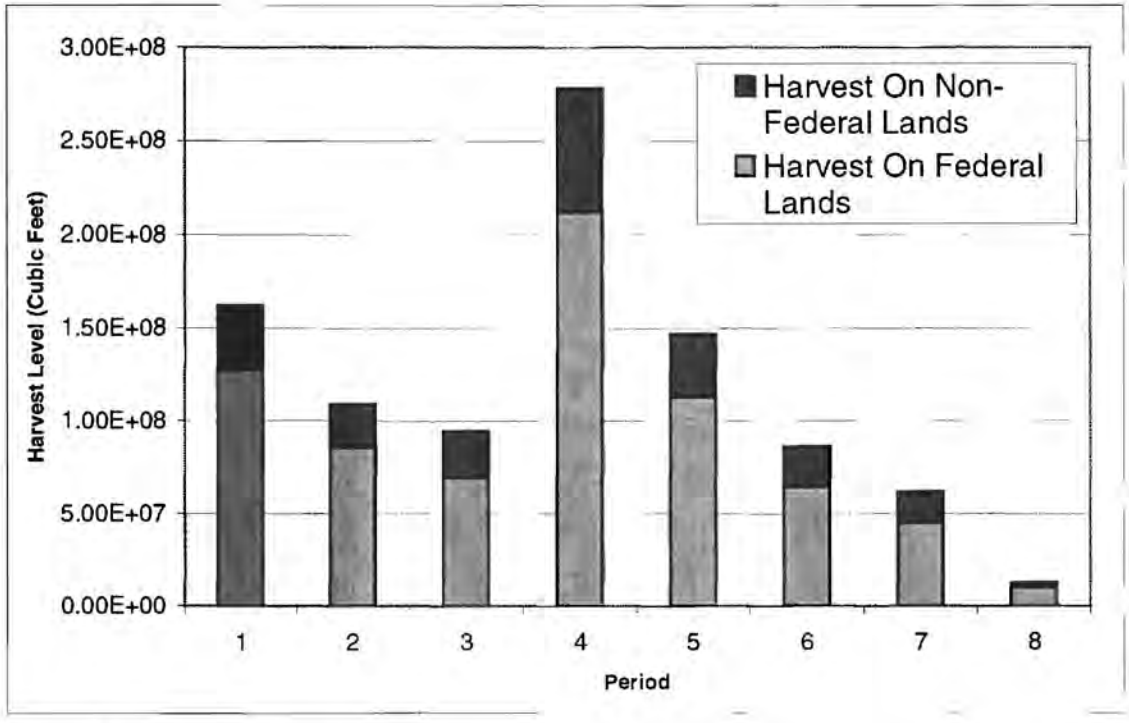

Figure 7-2: Before-simulation harvest levels in the Big Trees scenario

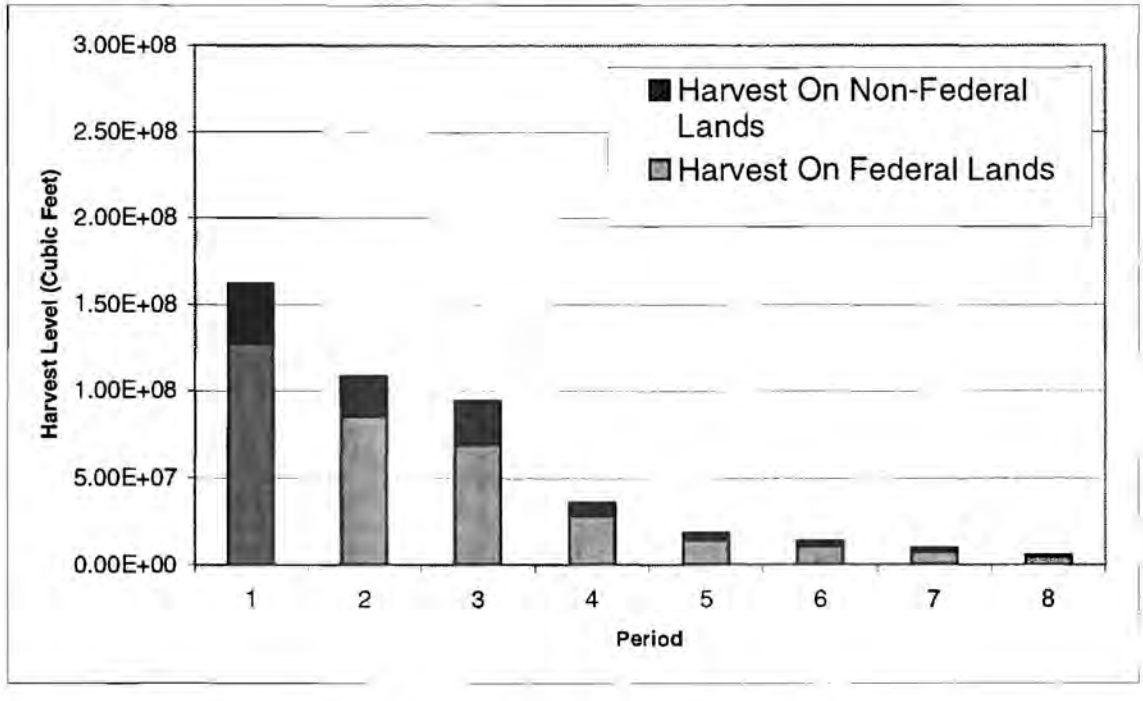

Figure 7-3: After-simulation harvest levels in the Big Trees scenario

The before-simulation results (Figure 7-2) show a significant harvest increase in period 4. This could probably be attributed to the prescription timing choices. Each prescription generated by PREMO in stage one has two timing choices for harvest activities: 1) harvest could occur immediately in period 1, or 2) harvest is not allowed for an additional 3 periods (i.e., period 4). The SafeD model currently does not track the prescription allocation by these two timing choices so I cannot determine the effect that the timing choice is having on the harvest levels.

The after-simulation result (i.e., episodic disturbance occurred) shows a rather different picture in terms of harvest levels. To understand something of what may be going on I must first restate that the only difference between the before- and after-simulation is that episodic disturbances and their effects have occurred. The two episodic disturbances included in the SafeD model are fire and insects. Fire occurs every period regardless of the weather pattern and insects occur only during periods with a drought-type weather pattern. As I will discuss in the results of the landscape simulation (stage three) there is a mild drought in period 3 and a severe drought in period 4. Any differences that are seen between the before- and after-simulation harvest level results are related to effects and adjustments caused by these episodic disturbances.

The before- and after-simulation harvest level values for periods 1, 2, and 3 will reflect only the fire disturbance. Episodic insect disturbance did occur in period 3 but harvest levels are calculated from PREMO after growth (for the period) and before any episodic disturbance. Therefore, the effects of any episodic disturbance in a given period will not be displayed until the following period. The fires encountered in periods 1, 2, and 3 were insignificant in size and this is reflected in the relatively unchanged before- and after-simulation harvest levels.

The after-simulation period 4 harvest level is significantly less than the before-simulation value. The before-simulation harvest level for period 4 is roughly 275 million cubic feet (MMCF). The after-simulation harvest level for period 4 drops to only 80 MMCF – a decrease of 195 MMCF. Recall that values displayed for period 4 reflect episodic disturbances that occurred in period 3. It is the occurrence of the first episodic insect disturbance event in period 3 that contributes to this dramatic decrease in harvest

levels. A severe drought occurs in period 4 and again a significant decrease in harvest levels are seen in period 5. There is no accounting for salvage harvest that occurs after an episodic disturbance. If there were the after-simulation harvest levels would be higher. The total acres affected by insects is so high (see tables in the following Landscape Simulation section), especially compared to the acres affected by fire, that it leaves little doubt about the cause-and-effect relationship between episodic insect disturbance and harvest levels in the Big Trees scenario: mortality caused by insect disturbance is going to greatly reduce harvest levels. Without accounting for episodic insect disturbance within the watershed any projections of harvest levels may be incorrect. Keep in mind that this is only one landscape goal evaluation (Big Trees scenario) and that other landscape scenarios may be developed which have alternative conclusions.

Landscape Simulation (stage three)

The simulation component of the SafeD model begins in stage three of the overall modeling strategy. Once a solution for a particular scenario is found in stage two that solution is stored and available for re-use in subsequent simulation runs.

To make a fair comparison of the results between the two scenarios all the stochastic elements (that SafeD generates) were held constant for each scenario. Those elements include:

1. The weather (influences insect and fire disturbances)
2. The number of fire ignition points per period (influences fire disturbance only)
3. The location of those ignition points (influences fire disturbance only)
4. The time and dates of fire burn (influences fire disturbance only)

Table 7-4 shows the weather pattern used for all simulation runs.

	Weather Pattern
Period 1	Moderate
Period 2	Moderate
Period 3	Mild Drought
Period 4	Severe Drought
Period 5	Moderate
Period 6	Wet
Period 7	Moderate
Period 8	Moderate

Table 7-4: Weather pattern used for both scenarios

Three statistics are compiled for each episodic disturbance that occurs during the simulation: 1) the acres affected by the disturbance, 2) the number of big trees killed by the disturbance (normalized to a per-acre value), and 3) the stand basal area (sq. ft.) that is killed by the disturbance (reflects all size classes). The episodic disturbances were not spatially limited to the landscape eligible for management (i.e., the modeled landscape) - they could occur in both managed and unmanaged areas.

Insects

The episodic insect disturbance is triggered by drought-type weather patterns. Only periods 3 and 4 are drought-type for the simulation runs. There are 15 years of growth before the first insect disturbance in period 3. The average number of big trees per-acre killed was much higher in both periods for the Big Trees scenario (Table 7-5) than the Grow Only scenario. This is probably due to there being more big trees available in the Big Trees scenario during both periods (Table 7-3) However, the basal area killed is nearly identical for both scenarios in both periods. There are over 8,000 more acres affected by insects in the Grow Only scenario during periods 3 and 4.

	Grow Only Scenario			Big Trees Scenario		
	Acres affected	# of big trees killed (per-acre)	Stand basal area killed (sf/acre)	Acres affected	# of big trees killed (per-acre)	Stand basal area killed (sf/acre.)
Period 3	414,084	.09	9.8	405,660	.19	10
Period 4	414,075	.05	17	404,293	.22	18

Table 7-5: Insect mortality statistics

In the Grow Only scenario stands (which were initially considered overstocked – see Introduction) accumulate their basal area from the smaller diameter classes because of the lack of growing space to create larger diameter trees. This would help explain the lower number of big trees killed – there are less available. On the other hand, the Big Trees scenario tried to manage the landscape so that there were big trees and thus, more should be available during insect disturbance.

The stand basal area killed in periods 3 and 4 support *a priori* expectations. Period 3, which had a mild drought weather pattern, had roughly 10 sq.ft./acre of basal area killed within each scenario. Referring to the modeling rules for thresholds and severity (Appendix C), this value "appears" consistent with the desired mortality (that is, the severity level applied per the rules). Period 4, a severe drought period, had roughly 18 sq.ft./acre of basal area killed within each scenario. Again, this appears consistent with the rules developed by the science team. Data needed to calculate actual harvest volume loss due to insects is currently not tracked by the SafeD model.

The forested acres effected by insects in both period 3 and 4 in both scenarios indicates that insects will play a major role in stand mortality. For the Grow Only scenario over 98% of the forested landscape is affected by insects. For the Big Trees scenario 96% of the forested landscape is affected in periods 3 and 4.

The rules for insect disturbance were developed to simulate expected losses over the long run (Agee, 1999). As to whether or not the implementation of these rules behaved in their predicted way I cannot determine. The science team members responsible for the insect modeling rules have not had a chance to thoroughly study these results. The implementation of the rules needs more review by the science team.

Fire

The number of fire ignition points and their location were constant for the two landscape scenarios and for the five simulation runs for each scenario (Table 7-6). However, each period within a simulation had different ignition points and locations.

	# of fire ignition points
Period 1	14
Period 2	6
Period 3	13
Period 4	10
Period 5	14
Period 6	12
Period 7	8
Period 8	6

Table 7-6: Number of ignition points for both scenarios

Fires occurred every period regardless of the weather pattern. The weather determined the wind, weather, and fuel moisture file FARSITE used (see discussion in Stage Three of The SafeD Model). Table 7-7 shows the fire statistics.

	Grow Only Scenario			Big Trees Scenario		
	Acres affected	# of big trees killed (per-acre)	Stand basal area killed (sf/acre.)	Acres affected	# of big trees killed (per-acre)	Stand basal area killed (sf/acre.)
Period 1	570	.27	115	1,028	.36	130
Period 2	504	.26	161	490	.33	167
Period 3	1,802	.08	168	1,842	.19	170
Period 4	563	.03	147	521	.05	144
Period 5	124	.02	146	91	.01	176
Period 6	92	.01	122	85	.02	132
Period 7	106	.01	132	67	.01	134
Period 8	113	.01	134	116	.03	137

Table 7-7: Fire mortality statistics

Only during period 3 did fires in both scenarios approach the Base-Case Fire mean area burned of 2,158 acres (the historical average). Otherwise the average acres burned per period in both scenarios fell below historic values. The fires were larger in the Big Trees scenario for periods 1,3, and 8.

The number of big trees killed (per-acre) was higher for the Big Trees scenario in all but periods 5 and 7. In period 5 the Grow Only scenario killed .02 big trees per-acre compared to .01 for the Big Trees scenario; and both were even at .01 big trees killed during period 7.

The average stand basal area killed was higher in all but period 4 in the Big Trees scenario. The greatest difference was in period 5 with the Grow Only scenario losing 146 sq. ft. of basal area compared to 176 sq. ft. in the Big Trees scenario.

The number of big trees killed by fire decreases significantly in both scenarios during periods 3 and 4. This corresponds with a significant drop in the PREMO calculation of big trees regardless of episodic disturbance (Table 7-2). Additionally, periods 3 and 4 are the two drought-type periods and encounter insect disturbance before the fires.

Evaluating the effectiveness of the episodic fire disturbance is based on two things:

1) Are the resulting fires and their associated flame lengths reasonable?
2) Do the fires approach the Base-Case Fire scenario (i.e., the historical record)?

Are the resulting fires and their associated flame lengths reasonable:

The flame lengths associated with all the fires (in all periods) for all the simulation runs made for this thesis were evaluated for "reasonableness". I used informal guidelines (Bahro, personal comm., Feb 2000) because no formal protocol has been established by the science team. In essence, I checked the flame length of each stand hit by a fire to see if it exceeded a threshold. The threshold, established at 20 feet, was used only to see if there were large number of stands that were exceeding that value. If a few stands exceeded the threshold there is not much of a concern because it is possible for flame lengths to reach that value. However, if I encountered many stands in many fires that were exceeding the threshold a flag was set to indicate that maybe the fire spread model was not operating properly or the inputs were not reasonable. There were no

incidences of the threshold being exceeded to a great extent in any fire during all the simulation runs.

Do the fires approach the Base-Case Fire scenario:

The Base-Case Fire is an attempt to parameterize past fire activity for describing the "best" scenario one could hope for in the future. The average acreage burned for a 5-year period in the Base-Case Fire scenario is 2,158 acres. That value is approached only in period 3 for both the Grow Only and Big Trees scenario. Period 3 had a mild drought weather pattern.

The reason for the disparity between the anticipated Base-Case Fire scenario and the results for these simulations is unclear at this time. The science team has not had the opportunity to complete an evaluation of the input fire parameters. This should be included in future work.

New Prescriptions

The number of new prescriptions needed for the Big Trees scenario is significantly greater than the Grow Only scenario for all periods except period 5 (Table 7-8). Period 8 is not shown because new prescriptions are not generated after period 8 – the simulation ends. The heterogeneity or homogeneity of the stand types affected by the episodic disturbance greatly affects the number of new prescriptions needed. In the Grow Only scenario there was only one stand prescription assigned to all the initial stands - grow only. The Big Trees scenario had a suite of 19 different prescriptions that were allocated across the landscape. Because the current prescription is one of the three variables that identify a new unique stand prescription I expected an increase in the number of new prescriptions needed for the Big Trees scenario.

The number of new prescriptions needed in periods 3 and 4 for the Big Trees scenario is significant. On the average PREMO can generate prescriptions at the rate of 5,000 per hour for prescriptions needing at least five periods worth of data. This translates into an approximate 4-hour process for PREMO to generate a single prescription for each of the new unique stand prescriptions needed in periods 3 and 4 for the Big Trees scenario.

	Grow Only Scenario		Big Trees Scenario	
	Total acres hit by insects and fire	# of new prescriptions needed	Total acres hit by insects and fire	# of new prescriptions needed
Period 1	570	242	1,028	2,984
Period 2	504	199	490	571
Period 3	415,886	2,415	407,502	20,594
Period 4	414,638	2,336	404,814	19,307
Period 5	124	133	91	22
Period 6	92	115	85	272
Period 7	106	111	67	112
Period 8	n/a	n/a	n/a	n/a

Table 7-8: New prescriptions needed as a result of episodic disturbances

DISCUSSION AND FUTURE WORK

At the beginning of the Sample Application section I outlined two purposes for the application I have discussed:

1. To exercise the SafeD model to see how it works.
2. To begin to understand the relationships modeled for the Applegate River Watershed.

These two purposes were in context for the goal of the Applegate Project:

> "To develop a forest landscape simulation model to use in evaluating the potential effects of different policies and forest management practices over time to achieve goals for the forest of the Applegate River Watershed in the context of possible stochastic events."

The sample application of the SafeD model demonstrates the development of a hybrid optimization/simulation model. It also demonstrates the need for further calibration and parameterization. However, that should not overshadow the accomplishment in creating the SafeD model.

The four-stage process designed for the SafeD model was a mixture of traditional modeling techniques and traditional strategic forest planning with some innovative strategies to overcome deficiencies or short-comings identified in these traditional methods. Stage one incorporates the widely-used FVS growth and yield model, but without the standard mortality equations, in a new prescription generator called PREMO. Stage two uses a heuristic programming technique, the Great Deluge Algorithm, to find a solution for a large landscape problem. Stages three and four incorporate the stochastic nature of episodic disturbances with processes for fire and insect disturbance events – then account for their effects and allowing management to react.

General Notes on SafeD Processes

Stand Delineation

The science team chose to break with traditional stand delineation (spatially defining polygons with similar vegetative or other attributes) and instead used each pixel from a classified Landsat TM image as a stand unit. This strategy was developed in part to allow the SafeD model to track the episodic fire disturbance at a fine-resolution (25 m. x 25 m.) and account for the effects of fires to specific stands. This is in contrast to applying fire disturbance effects to larger spatial units on a distribution approach (e.g., see discussion on the SAFE FOREST model in the Literature Review). This strategy has some drawbacks which should be addressed.

Allowing each 25 meter pixel to represent a stand is difficult to implement from a tactical standpoint. It is unlikely that if one stand is different from its surrounding stands (for this example we assume the surrounding stands are identical) that the single stand would actually be treated or managed uniquely. But this must be weighed with the fact that it is possible for this single stand to react differently to episodic disturbances such as fire or insects. After running the simulations presented in this thesis I have come to two seemingly contradictory conclusions about the stand delineation strategy we used: 1) that the use of 25 meter pixels as stands may be ineffective for use when confronted with a large landscape-level problem, and 2) that the use of 25 meter pixels as stands enhances the ability to account for the effects of episodic disturbances.

I have discussed the results of the GDA heuristic I used to solve the landscape problem associated with the Big Trees scenario. By using 25 meter pixels as stands the problem has over 2.2 million decision variables for the stands alone. The potential number of unique combinations of stands to prescriptions is $19^{2.2\ \text{million}}$. The GDA evaluated only 29 million of those combinations. After running these simulations I am left with the feeling that another approach to stand delineation may be more suitable in regards to solving a landscape-level problem. The CLAMS project (Bettinger et al., 2000a) is using a strategy that aggregates Landsat TM pixels into larger basic simulation units (BSU) based on vegetation, stand structure, and other attributes. These BSU's are

then combined (using unique intersections of an additional set of attributes) to form even larger management "parcels". These parcels serve as the management decision unit. However, the attributes of the individual BSU's that make up a parcel are maintained and tracked during the simulation. I feel a similar strategy that aggregates data for management decisions but maintains finer resolution data for episodic disturbances may prove useful for the SafeD model in future work.

However, the use of 25 meter pixels as stands was essential to our efforts to apply episodic disturbance on the landscape. The FARSITE fire spread model will accept input landscape data at any resolution (i.e., the size of the pixels). By delineating stands at 25 meters we were able to provide a detailed description of the fuel model and other necessary attributes for FARSITE at the pixel level. These attributes reflected the stand more accurately in that they were not calculated as a percentage of some larger stand unit.

Future efforts on the SafeD model should explore other options for stand delineation. I would hesitate to state that using 25 meter pixels as stand units is impossible; I have shown that it can be done. I would only suggest that the usefulness of maintaining such fine-resolution stand data may prove futile with large landscape problems.

Variability Assessment

The results seen in this thesis are from a single simulation run of both the Grow Only scenario and the Big Trees scenario. I did make five simulations runs of each but not all the data was maintained because they were done while I was finishing "checking-out" that the model worked. The issues surrounding variability of the results are important. Trends can be seen with multiple simulations that may not appear with a single simulation. Future work on the SafeD model should incorporate a more thorough variability assessment.

Prescription Generation (stage one)

PREMO

There are some limitations and considerations in regards to PREMO; both in the theory behind PREMO achieving stand optimality and in the implementation of PREMO with regards to this thesis. See Wedin (1999) for her discussion on the theory of PREMO and its ability to achieve stand optimality. This thesis is concerned with the implementation of PREMO.

PREMO was designed to generate a suite of goal-oriented optimized prescriptions for the Applegate River Watershed. It has an automated approach in that the goals are pre-programmed, the growth and yield equations are pre-programmed (for known vegetation in the watershed), and the optimization search procedures are also pre-programmed. This is a departure from traditional prescription generation in which a modeler works with a growth and yield model and through manual trial-and-error to develop good prescriptions. The PREMO approach has two distinct advantages: 1) it has the ability to develop "optimal" prescriptions if a stand goal has been identified and variables are in place to measure attainment of optimality; and 2) it has the ability to create these new prescriptions "on-the-fly" during a simulation – they do not have to be created *a priori*.

The SafeD model takes full-advantage of PREMO. The suite of prescriptions generated by PREMO in stage one are optimized for specific stand goals for each of the existing stands on the watershed. Having this suite of prescriptions to explore allows the landscape optimization component of the SafeD model to work effectively. PREMO can generate the prescriptions for the initial landscape in only 37 minutes – that's over 2,500 prescriptions. Any change in the weights or variables used within PREMO can be accomplished with minimal effect on the total simulation time. If traditional methods are used to generate optimal prescriptions it would not be unreasonable to expect delays of days or even weeks to incorporate such changes. However, future work on the SafeD model should include making the PREMO model more efficient.

The ability for PREMO to generate on-the-fly prescriptions proved crucial to the development of the SafeD model. The behavior, extent, and effect of stochastic episodic disturbances cannot be predicted.. The idea of trying to develop a suite of prescriptions *a priori* that would account for any episodic disturbance, in any period, for any combination of disturbance type and severity is unreasonable for the spatial resolution we are using for the Applegate Project. For example, ultimately there were over 43,000 new prescriptions needed for the Big Trees scenario over the entire planning horizon (see Table 7-8). It would have been nearly impossible for the science team to predict which new prescriptions were needed during the simulation. Therefore, a suite of millions of prescriptions would be needed. In the end, only the 43,000 were used. The time and effort saved by not creating a suite of millions of prescriptions should certainly be considered a benefit.

The issues surrounding my discussion on mortality in PREMO are unresolved. Future work should also include ensuring that PREMO behaves in a predictable way.

Landscape Optimization (stage two)

The landscape optimization (stage two) required the use of a heuristic technique to obtain a solution to the landscape problem presented. The large number of integer variables and the complexity of the evaluation procedures make locating an optimal solution with traditional mathematical programming techniques (e.g., integer programming) a computationally difficult process. A comparison of the heuristic algorithm used for this thesis with other algorithms is not presented because no existing algorithm uses the same evaluation procedures for the landscape goal presented in this thesis.

I have demonstrated the use of the Great Deluge Algorithm in obtaining a solution for a large landscape in which the problem is to simultaneously meet a landscape goal subject to a spatial constraint. I ensure that the land management activities are compatible with the landscape goal by always staying in the feasible region of possible solutions.

Future work should incorporate processes to address the deficiencies I noted regarding the ability to measure the success of the landscape optimization. This includes: 1) obtaining multiple solutions, 2) more effort on calibration, 3) ensuring the Objective Function is sufficient for the problem, and 3) ensure the quality of the input stand data.

Landscape Simulation (stage three and four)

Episodic Disturbances (stage three)

Stage three of the SafeD model is the landscape simulation. The unique contribution of the SafeD model is the incorporation of episodic disturbances (insects and fire). Furthermore, the spatial resolution used (25 m. x 25 m.) to initiate, distribute, and track the episodic disturbances (and effects) demonstrates an improved strategy over traditional methods of applying disturbance effects on a distributional approach.

The results seen from the simulations presented in this thesis present three interesting conclusions which may have significant implications for forest management practices and polices in the Applegate River Watershed. First, this thesis has shown that to achieve a landscape goal of maximizing the number of big trees across the landscape timber harvesting will need to occur. The Grow Only scenario had significantly less big trees than the Big Trees scenario (which included timber harvesting) both in the before- and after-simulation analysis. Secondly, the effects of episodic insect disturbance is going to negate the need for as much timber harvesting that would be projected without accounting for such disturbances. And third, fire is going to play a significantly less role, in regards to tree mortality, than insects will.

The first conclusion, that timber harvest needs to occur to achieve a landscape goal of big trees, is somewhat expected. There are currently high stand densities in the watershed. Those densities must be reduced to achieve diameter growth. Therefore, the forests in the watershed need to be managed. Management can come from one of two ways: 1) timber harvesting, or 2) episodic disturbances. If we ignore episodic disturbances for the moment, then the results seen in this thesis demonstrate that management with timber harvesting will produce more big trees than a "hands-off"

approach. However, if we consider episodic disturbances then the other two conclusions I stated come into play.

The second conclusion, that episodic insect disturbance is going to negate the need for timber harvesting in the Big Trees scenario, is seen in Figures 7-2 and 7-3. The after-simulation harvest levels dropped significantly in the periods following the first episodic insect disturbance (period 3). However, there are still more big trees in the Big Trees scenario, which allowed timber harvesting, than in the Grow Only scenario. The occurrence of the episodic insects in periods 3 and 4 greatly reduced the harvest levels. It should be noted that the SafeD model has no salvage prescription to account for those trees killed by insects and harvested while they are still useful (that capability would increase the harvest level values).

The third conclusion, that fire is going to play a significantly less role in tree mortality, is seen by comparing Tables 7-5 and 7-7. In either scenario the fires affected less than 5,000 total acres during the entire 40-year planning horizon. The insects affected over 400,000 acres during each of the two periods they occurred. This is a significant finding. Current management policies in the watershed are directed towards fire reduction. These results are showing that fire is insignificant compared to the mortality insects will cause.

Future work to the SafeD model should include more evaluation of the episodic disturbance processes. There are two questions that should be addressed: 1) are the episodic disturbances doing what we want them to do in regards to mortality; and 2) is this method of representing mortality through episodic disturbances really a viable alternative?

Re-Optimization (stage four)

I discussed a modified strategy that we chose for stage four at the end of the discussion in The SafeD model. In essence, the idea of re-optimizing the selection of stand prescriptions proved too complicated for implementation at this time. The simpler strategy used for this thesis makes an assumption that should be addressed in future work; that the same prescription allocation should be prescribed to a stand after the occurrence

of any episodic disturbance on that stand. It may be more useful to evaluate the stand from a different view and consider "now that this stand has been affected by an episodic disturbance, what do I want to do to it in relation to other management considerations?"

CONCLUDING REMARKS

The goal of developing a forest landscape simulation model to use in evaluating the potential effects of different policies and forest management practices over time while achieving goals for the forest in the context of stochastic events has been met with the SafeD model. However, the SafeD model is in its infancy and this thesis presents only a single application – revealing some accomplishments and deficiencies in both the ideas behind the model and the parameters and processes used within the model.

The focus of this thesis has been to highlight my contribution to the Applegate Project: the development of a hybrid landscape optimization/simulation model that incorporates episodic disturbance at a fine-resolution - the SafeD model. This is an original contribution to the landscape modeling field. Traditional landscape optimization and landscape simulation models have worked on parallel tracks, often in isolation of one another. The SAFE FOREST model (Sessions, 1999) is the predecessor to the SafeD model in bridging the gap between optimization and simulation models. The optimization component (stage two) of the SafeD model is an improvement in traditional landscape optimization for two reasons: 1) the linkage to the simulation component of the model, and 2) the use of heuristic programming techniques to solve a large landscape problem. Furthermore, the SafeD model is an improvement over traditional landscape simulation models for two reasons: 1) the maintenance of fine spatial resolution, and 2) the incorporation of episodic disturbances, particularly insects. Additionally, by maintaining a fine-resolution of data we are able to use the well-documented and widely-used fire spread model called FARSITE (Finney, 1998).

In the Literature Review section I showed three antecedent landscape models in a comparison matrix. Table 9-1 shows the same matrix with the addition of the SafeD model. Although there is more work to do in testing and development of the SafeD model it does meet the goals laid out in the matrix. Success of the SafeD model should be judged on the successful completion in meeting these goals. In this regard I feel the development and sample application of the SafeD model has been a success.

		SAFE FOREST	LANDIS	CLAMS	SafeD
Spatial Data Components	Study area size	1 million acres	1.5 million acres	5 million acres with 8 "megasheds"	**493,000 acres**
	Data structure	vector	raster	vector	**raster**
	Resolution (MMU)	varies	200 meter x 200 meter	varies	**25 meter x 25 meter**
Desirable Model Characteristics Component	Recognize economical and ecological	both	ecological	both	**both**
	Optimize multiple goals	yes	no	no	**yes**
	Represent forest management activities	yes	yes	yes	**yes**
	Represent Stochastic elements	yes	yes	no	**yes**
	Represent FIRE - Spatially Explicit	"partial"	yes	no	**yes**
	Represent INSECTS - Spatially Explicit	no	no	no	**yes**
	Repeated simulations to assess variability	yes	yes	yes	**yes**

Table 9-1: Comparison matrix for recent landscape analysis models and SafeD

REFERENCES

Agee, Jim. 1999. Rules for potential vegetation; Rules for regeneration; Rules for disturbance; FOFEM tables; Fire and fuel inputs; A mimic of natural fire. September 17th document outlining inputs to the SafeD model. Unpublished.

Anderson, H.E.. 1982. Aids to determining fuels for estimating fire behavior. Gen. Tech. Report INT-122. U.S.Dept. of Agriculture, Forest Service, Rocky Mountain Research Station.

Bettinger, Pete, J. Sessions, and K. Boston. 1997. Using Tabu search to schedule timber harvests subject to spatial wildlife goals for big game. Ecological Modelling. 94:111-123.

Bettinger, Pete, J. Sessions, and K.N. Johnson. 1998. Ensuring the compatibility of aquatic habitat and commodity production goals in eastern Oregon with a tabu search procedure. Forest Science. 44(1):96-112.

Bettinger, Pete, J.Sessions, T. Spies, J. Brooks, and A. Herstrom. 2000a. Landscape Simulation Model for Coastal Oregon Landscape Analysis and Modeling. Paper 3379 of Forest Research Laboratory, Oregon State University. (in review) Forest Science.

Bettinger, Pete, D. Graetz, K. Boston, J. Sessions, W. Chung. 2000b. Eight heuristic planning techniques applied to three increasingly difficult wildlife planning problems. (in review) Forest Science. May 2000. 69pp..

Beukema, S.J., E. Reinhardt, J.A. Greenough, W.A. Kurz, N. Crookston, and D.C.E. Robinson. 1998. Fire and fuels extension: model description, working draft. Prepared by ESSA Technologies Ltd., Vancouver, BC for USDA Forest Service, Rocky Mountain Research Station, Missoula, MT. 58 pp..

Boychuk, D., and D. Martell. 1996. A multistage stochastic programming model for sustainable forest-level timber supply under risk of fire. Forest Science. 42(1):10-26.

Carlson, Joan and C. Christiansen. 1993. Eldorado National Forest: Cumulative off-site watershed effects (CWE) analysis process – Draft Version 1.1. Eldorado National Forest, Supervisor's Office. Placerville, CA.

Clements, Stephen E., P.L. Dallain, and M.S. Jamnick. 1990. An operational, spatially constrained harvest scheduling model. Canadian Journal of Forest Research. 20:1438-1447.

Daust, David K., and J.D. Nelson. 1993. Spatial reduction factors for strata-based harvest schedules. Forest Science. 39(1):152-165.

Davis, L.S. and K.N Johnson. 1987. Forest management. Third edition. McGraw-Hill, New York. 790 pp..

Dixon, Gary and Ralph Johnson. 1995. The Klamath Mountains geographic variant of the Forest Vegetation Simulator Version 6.1. USDA Forest Service, Washington Office, Forest Management Service Center, Fort Collins, CO. 19p. FMSC Internal Report.

Dowsland, Kathryn A. 1993. Simulated Annealing. P.20-69 *in* Modern heuristic techniques for combinatorial problems. Reeves, C.R. (ed.). John Wiley & Sons, Inc., New York.

Dueck, Gunter. 1993. New optimization heuristics – The great deluge algorithm and the Record-to-Record travel. Journal of Computational Physics. 104:86-92.

Finney, Mark A. 1998. FARSITE: Fire Area Simulator-model development and evaluation. Res.Pap. RMRS-RP-4, Ogden, UT: U.S.Dept. of Agriculture, Forest Service, Rocky Mountain Research Station. 47 pp..

Finney, Mark A.. 1999. FLAMMAP Model. Unpublished description. Systems for Environmental Management. Missoula, MT.

Frelich, L.E. and C.G. Lorimer. 1991. Natural disturbance regimes in hemlock-hardwood forests of the upper Great Lakes region. Ecological Monographs. 61:159-162.

Glover, Frank, and M. Laguna. 1993. Tabu search. P.70-150 *in* Modern heuristic techniques for combinatorial problems. Reeves, C.R. (ed.). John Wiley & Sons, Inc., New York.

Golley, F.B. 1993. Development of landscape ecology and its relation to environmental management. In *Eastside forest ecosystem health assessment. Volume II. Ecosystem management: principles and applications.* M.E. Jensen and P.S. Bourgeron (ed.). pp. 37-44. USDA Forest Service, Missoula, MT, USA.

Hof, J.G., K.S. Robinson, and D.R. Betters. 1988. Optimization with expected values of random yield coefficients in renewable resource linear programs. Forest Science. 34(3):634-646.

Hof, J.G, and L. Joyce. 1992. Spatial optimization for wildlife and timber in managed forest ecosystems. Forest Science. 38:489-508.

Hof, J.G., M. Bevers, L. Joyce, B. Kent. 1994. An integer programming approach for spatially and temporally optimizing wildlife populations. Forest Science. 40:177-191.

Hoganson, H.M., and D. Rose. 1984. A simulation approach for optimal timber management scheduling. Forest Science. 30(1):220-238.

Hoganson, H.M. and T.E. Burk. 1997. Models as tools for forest management planning. Commonwealth Forestry Review. 76:11-17.

Iverson, D.C. and R.M. Alston. 1986. The genesis of FORPLAN: A historical and analytical review of Forest Service planning models. USDA Forest Service, General Technical Report INT-214.

Johnson, K.N., and H.L. Scheurman. 1977. Techniques for prescribing optimal timber harvest and investment under different objectives – discussion and synthesis. Forest Science Monograph 18. Washington DC: Socienty of American Foresters.

Johnson, K.N., D.B. Jones, and B.M. Kent. 1980. Forest Planning Model (FORPLAN). User's Guide and Operations Manual. USDA Forest Service, Fort Collins, Co. 251 pp..

Johnson, K.N. 1992. Consideration of watersheds in long-term forest planning models: The case of FORPLAN and its use on the national forest. In *Watershed management: Balancing sustainability and environmental change*. R.J. Naiman (ed.). pp. 347-360. Springer-Verlag: New York, NY.

Johnson, K.N., J. Sessions, J. Franklin, and J. Gabriel. 1998. Integrating wildfire into strategic planning for Sierra Nevada forests. Journal of Forestry. 96(1):42-49.

Kessell, S.R.. 1979. Gradient modeling: resource and fire management. Springer-Verlag, New York.

Lockwood, Carey, and T. Moore. 1993. Harvest scheduling with spatial constraints: a simulated annealing approach. Canadian Journal of Forest Research. 23(3):468-478.

Mellen, K., and A. Ager. 1998. Coarse Woody Debris Model - Version 1.2. USDA Forest Service, Mt. Hood and Gifford Pinchot National Forest.

Mladenoff, David and W. Baker. 1999. Development of forest and landscape modeling approaches. Pp. 1-13 in "Spatial Modeling of Forest Landscape Change: approaches and applications",. Mladenoff, David and W. Baker (Eds). Cambridge University Press, UK..

Mladenoff, David and H.S. He. 1999. Design, behavior and application of LANDIS, an object-oriented model of forest landscape disturbance and succession. Pp. 125-162 in "Spatial Modeling of Forest Landscape Change: approaches and applications". Mladenoff, David and W. Baker (Eds). Cambridge University Press, UK..

Murray, Alan T., and R.L. Church. 1995. Heuristic solution approaches to operational forest planning problems. OR Spektrum [Operations Research]. 17:193-203.

Nelson, John, and G. Liu. 1994. Scheduling cut blocks with simulated annealing. Canadian Journal of Forest Research. 24(2): 365-372.

O'Hara, A.J., B.A. Faaland, and B.B. Bare. 1989. Spatially constrained timber harvest scheduling. Canadian Journal of Forest Research. 19:715-724.

Reed, W.J. and D. Enrico. 1986. Optimal harvest scheduling at the forest level in the presence of the risk of fire. Canadian Journal of Forestry Research. 16: 266-278.

Reeves, Colin R. 1993. Modern heuristic techniques for combinatorial problems. Editor. John Wiley & Sons, Inc., New York.

Reinhardt, Elizabeth D., R. Keane, and J. Brown. 1997. First Order Fire Effects Model: FOFEM 4.0, user's guide. GTR, INT-GTR-344. Ogden, UT: U.S. Dept. of Agriculture, Forest Service, Intermountain Research Station. 65 p..

Rothermel, Richard C. 1972. A Mathematical Model for Predicting Fire Spread in Wildland Fuels. Research Paper. INT-115. Ogden, UT: U.S. Dept. of Agriculture. Forest Service, Intermountain Forest and Range Experiment Station.

Runkle, J.R.. 1982. Patterns of disturbance in some old-growth mesic forest of eastern North America. Ecology. 63:1533-1546.

Sessions, John, K.N. Johnson, J. Franklin, and J. Gabriel. 1999. Achieving sustainable forest structures on fire-prove landscapes while pursuing multiple goals. Pp. 210-255 in "Spatial Modeling of Forest Landscape Change: approaches and applications", Mladenoff, David and W. Baker (Eds.). Cambridge University Press, UK..

(SNEP) Sierra Nevada Ecosystem Project. 1996. *Status of the Sierra Nevada: Final Report to Congress by the Sierra Nevada Ecosystem Project (SNEP),* Wildland Resource Center Report No. 36, 3 volumes, University of California, Davis, Calif..

USDA Forest Service and USDI Bureau of Land Management. 1994. Record of Decision for Amendments to Forest Service and Bureau of Land Management Planning Documents Within the Range of the Northern Spotted Owl; Standards and Guidelines for Management of Habitat for Late-Successional and Old-Growth Forest Related Species Within the Range of the Northern Spotted Owl. Washington, DC: U. S. Government Printing Office.

USDI Bureau of Land Management, Medford District, USDA Forest Service, Rogue River National Forest, USDA Forest Service, Siskiyou National Forest, USDA Forest Service, PNW Research Station. 1994. Applegate Adaptive Management Area Ecosystem Health Assessment. Pp. 1-76.

Van Wagner, C.E.. 1969. A simple fire growth model. Forestry Chronicl. 45:103-4.

Van Wagner, C.E.. 1978. Age class distribution and the forest fire cycle. Canadian Journal of Forest Research. 8:220-7.

Voß, Stefan. 1993. Tabu search: applications and prospects. Pp. 333-353 *in* Network optimization problems, Du, D.Z., and P.M. Pardalos (eds.). World Scientific Publishing Co., Singapore.

Wedin, Heidi. 1999. Stand Level Prescription Generation under Multiple Objectives. M.S. Thesis. Oregon State University, Corvallis, OR. 178 p.

Yoshimoto, A., R.G. Haight, and J.D. Brodie. 1990. A comparison of the pattern search algorithm and the modified PATH algorithm for optimizing an individual tree model. Forest Science. 36:394-412.

Zanakis, S.H., and J.R. Evans. 1981. Heuristic "optimization": why, when, and how to use it. Interfaces. 11(5):84-89.

APPENDICES

Appendix A: Vegetation and Structural Stage Classification

Stage	Barren	Water	Shrub	Grass/ Forbs	Red fir	Mixed conifer < 3000'	White fir	Pine	Closed cone pine	Deciduous hardwood	Conifer hardwood	Evergreen hardwood	Mixed conifer > 3000'	Totals
Barren	7,168													7,168
Water		1,565												1,565
Shrub			30,345											30,345
Grass/ Forbs				32,473										32,473
0 - 4.9" all cc %					571	1,087		1,952	906	4,568	4,928	7,808	1,662	23,482
5 - 8.9" < 60% cc						1,083		3,710	373	5,374	18,560	1,588	3,816	34,504
5 - 8.9" >= 60% cc								438	369					807
9 - 14.9" < 60% cc						781	1,298	10,352		20,493	14,253	883	350	48,410
9 - 14.9" >= 60% cc						8,842		15,054		7,122	50,879	17,772	8,022	107,691
15 - 20.9" < 60% cc					1,710	13	1,070	4,354		3,685	2,416	72	1,797	15,117
15 - 20.9" >= 60% cc					3,193	30,330	4,732			3,401	39,803	5,522	27,924	114,905
21 - 24.9" < 60% cc					850	660	2,594	928					644	5,676
21 - 24.9" >= 60% cc					3,199	318	4,776				6,311		3,870	18,474
25 - 31.9" all cc %					1,934	3,971	12,494	144					27,569	46,112
32"+ all cc %						21	342						5,571	5,934
Totals	7,168	1,565	30,345	32,473	11,457	47,106	27,306	36,932	1,648	44,643	137,150	33,645	81,225	492,663

Table 1: Acres in each vegetation - structural stage class

Appendix B: Plant Association Group (PAG) Assignment Rules

Plant Association Groups (PAGs)

1. Douglas-fir/Dry
2. Douglas-fir/Wet
3. White fir/Dry
4. White fir/Wet
5. Red fir
6. Jeffrey pine
7. Pine/oak

Codes used:

Ppt	= precipitation in inches
Elevation	= values in feet
Slope	= values in %
Aspect	= values in degrees

Rules for Assigning PAGs to Each Stand

	PAG
Geology serpentine	Jeffrey pine
Other	Continue
1. Elevation ≤ 2000	
2A. Slope ≤ 15%	Pine/oak
2B. Slope > 15%	Continue to 3
3A. Aspect 135-225	Pine/oak
3B. Aspect other	Douglas-fir/Dry
4. Elevation > 2000 and ≤ 2500	
5A. Ppt ≤ 35	Pine/oak
5B. Ppt. > 35	
6A. Ppt ≤ 40	Douglas-fir/Dry
6B. Ppt. > 40	Douglas-fir/Wet
7. Elevation > 2500 and ≤ 3500	
8A. Ppt ≤ 40	Douglas-fir/Dry
8B. Ppt > 40	
9A. Aspect 1-45 and 316-360	White fir/Dry
9B. Aspect Other	
10A. Aspect 226-315 and 46-135	Douglas-fir/Wet
10B. Aspect 136-225	Douglas-fir/Dry

11. Elevation > 3500 and ≤ 4000
    - 12A. Aspect 271-360 and 1-90 . . . Douglas-fir/Wet
    - 12B. Aspect 91-270 . . . . . White fir/Dry

13. Elevation > 4000 and ≤ 4500
    - 14A. Ppt. ≤ 45 . . . . . Douglas-fir/Wet
    - 14B. Ppt. > 45
        - 15A. Aspect 136-225 . . . . Douglas-fir/Wet
        - 15B. Aspect 46-135 and 226-315 . . White fir/Dry
        - 15C. Aspect 316-360 and 1-45 . . White fir/Wet

16. Elevation > 4500 and ≤ 5000
    - 17A. Ppt ≤ 50
        - 18A. Aspect 136-225 . . . . Douglas-fir/Wet
        - 18B. Aspect 91-135 and 226-270 . . White fir/Dry
        - 18C. Aspect 271-360 and 1-90 . . White fir/Wet
    - 17B. Ppt > 50
        - 19A. Aspect 136-225 . . . . White fir/Dry
        - 19B. Aspect 46-135 and 226-315 . . White fir/Wet
        - 19C. Aspect 316-360 and 1-45 . . Red fir

20. Elevation > 5000 and ≤ 5500
    - 21A. Ppt ≤ 50
        - 22A. Aspect 158-202 . . . . Douglas-fir/Wet
        - 22B. Aspect 136-157 and 203-225 . . White fir/Dry
        - 22C. Aspect 226-360 and 1-135 . . White fir-Wet
    - 21B. Ppt > 50 and ≤ 60
        - 23A. Aspect 46-315 . . . . White fir/Wet
        - 23B. Aspect 316-360 and 1-45 . . Red fir
    - 21C. Ppt > 60
        - 24A. Aspect 271-360 and 1-90. . . Red fir
        - 24B. Aspect 91-270 . . . . White fir/Wet

25. Elevation > 5500 and ≤ 6000
    - 26A. Ppt ≤ 60
        - 27A. Aspect 136-225 . . . . White fir/Wet
        - 27B. Aspect 1-135 and 226-360 . . Red fir
    - 26B. Ppt > 60 . . . . . Red fir

28. Elevation > 6000. . . . . Red fir

Appendix C: Insect Disturbance Rules

DOUGLAS-FIR KEY

Thresholds of basal area (ba) per PAG

White fir/dry and Douglas-fir/wet:	> 250 sq ft/ac.
Douglas-fir/dry:	> 120 sq ft/ac.
Pine/Oak:	> 80 sq ft/ac.

Severity (applied to treelist)

Mild drought:	10% of ba of Douglas-fir killed (>10" DBH)
Severe drought:	20% of ba of Douglas-fir killed (>10" DBH)

TRUE FIR KEY

Thresholds of basal area (ba) per PAG

Red fir or White fir/wet:	> 250 sq ft/ac.
White fir/dry:	> 120 sq ft/ac.
Douglas-fir (wet or dry):	any
Pine/Oak:	any

Severity (applied to treelist)

Red fir series:

Mild drought:	10% of ba of White and Red fir killed (all sizes)
Severe drought:	20% of ba of White and Red fir killed (all sizes)

White fir series:

Mild drought:	10% of ba of White fir killed (all sizes)
Severe drought:	20% of ba of White fir killed (all sizes)

Douglas-fir series:

Mild drought:	20% of ba of White fir killed (all sizes)
Severe drought:	40% of ba of White fir killed (all sizes)

Pine/Oak:

Mild drought:	40% of ba of White fir killed (all sizes)
Severe drought:	60% of ba of White fir killed (all sizes)

PINES KEY

Thresholds of basal area (ba) per PAG

Jeffrey pine and Pine/Oak:	> 80 sq ft/ac.
Douglas-fir/dry and White fir/dry:	> 120 sq ft/ac.
Douglas-fir/wet, White fir/wet, and Red fir:	> 180 sq ft/ac.

Severity (applied to treelist)

Mild drought:	10% of ba of all pines killed, largest first
Severe drought:	30% of ba of all pines killed, largest first

Appendix D: FOFEM Tables

Each of the six tables below represent the FOFEM mortality index used for the particular species indicated. The columns are broken into two foot flamelength intervals. The rows are DBH intervals. The table value is a percentage such that a value of 0.65 means that 65% of the trees-per-acre are killed.

	Flamelength Category							
DBH Category	2	4	6	8	10	12	14	16
1	1	1	1	1	1	1	1	1
2	0.9	1	1	1	1	1	1	1
4	0.85	0.95	1	1	1	1	1	1
6	0.75	0.95	1	1	1	1	1	1
8	0.65	0.85	1	1	1	1	1	1
10	0.45	0.7	0.95	1	1	1	1	1
12	0.4	0.65	0.9	1	1	1	1	1
14	0.35	0.6	0.8	1	1	1	1	1
16	0.3	0.55	0.75	1	1	1	1	1
18	0.25	0.5	0.75	0.95	1	1	1	1
20	0.2	0.45	0.65	0.85	1	1	1	1
22	0.2	0.4	0.65	0.85	0.95	1	1	1
24	0.2	0.35	0.55	0.85	0.95	1	1	1
26	0.2	0.3	0.55	0.75	0.95	1	1	1
28	0.15	0.25	0.5	0.75	0.95	1	1	1
30	0.1	0.25	0.45	0.75	0.9	1	1	1
32	0.1	0.25	0.45	0.75	0.9	1	1	1
34	0.1	0.25	0.45	0.65	0.85	1	1	1
36	0.1	0.2	0.35	0.65	0.8	1	1	1
38	0.1	0.2	0.35	0.55	0.75	1	1	1
40	0.1	0.2	0.35	0.55	0.75	0.9	1	1

Table 2: Oregon white oak

DBH Category	Flamelength Category							
	2	4	6	8	10	12	14	16
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
4	0.6	1	1	1	1	1	1	1
6	0.5	0.9	1	1	1	1	1	1
8	0.3	0.6	1	1	1	1	1	1
10	0.3	0.3	0.9	1	1	1	1	1
12	0.2	0.2	0.8	1	1	1	1	1
14	0.1	0.1	0.6	1	1	1	1	1
16	0.1	0.1	0.4	0.9	1	1	1	1
18	0.1	0.1	0.2	0.9	0.9	0.9	0.9	0.9
20	0.1	0.1	0.2	0.7	0.9	0.9	0.9	0.9
22	0.1	0.1	0.1	0.6	0.9	0.9	0.9	0.9
24	0	0	0.1	0.5	0.9	0.9	0.9	0.9
26	0	0	0.1	0.4	0.8	0.9	0.9	0.9
28	0	0	0	0.3	0.8	0.9	0.9	0.9
30	0	0	0	0.2	0.7	0.9	0.9	0.9
32	0	0	0	0.1	0.6	0.9	0.9	0.9
34	0	0	0	0.1	0.6	0.8	0.8	0.8
36	0	0	0	0.1	0.5	0.8	0.8	0.8
38	0	0	0	0.1	0.4	0.8	0.8	0.8
40	0	0	0	0.1	0.3	0.7	0.8	0.8

Table 3: Douglas fir

DBH Category	Flamelength Category							
	2	4	6	8	10	12	14	16
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
4	0.95	1	1	1	1	1	1	1
6	0.75	0.95	1	1	1	1	1	1
8	0.7	0.95	1	1	1	1	1	1
10	0.7	0.9	1	1	1	1	1	1
12	0.6	0.85	1	1	1	1	1	1
14	0.6	0.8	0.95	1	1	1	1	1
16	0.55	0.75	0.95	1	1	1	1	1
18	0.5	0.65	0.9	1	1	1	1	1
20	0.5	0.65	0.85	1	1	1	1	1
22	0.4	0.65	0.85	1	1	1	1	1
24	0.4	0.65	0.85	1	1	1	1	1
26	0.35	0.65	0.85	0.95	1	1	1	1
28	0.3	0.65	0.85	0.95	1	1	1	1
30	0.3	0.65	0.85	0.95	1	1	1	1
32	0.25	0.4	0.55	0.9	1	1	1	1
34	0.25	0.4	0.55	0.8	1	1	1	1
36	0.25	0.35	0.55	0.8	1	1	1	1
38	0.25	0.3	0.45	0.7	1	1	1	1
40	0.25	0.3	0.45	0.65	0.95	1	1	1

Table 4: Other hardwoods

DBH Category	Flamelength Category							
	2	4	6	8	10	12	14	16
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
4	0.7	1	1	1	1	1	1	1
6	0.6	0.6	1	1	1	1	1	1
8	0.4	0.4	1	1	1	1	1	1
10	0.3	0.3	0.7	1	1	1	1	1
12	0.2	0.2	0.3	1	1	1	1	1
14	0.2	0.2	0.2	1	1	1	1	1
16	0.1	0.1	0.1	0.8	1	1	1	1
18	0.1	0.1	0.1	0.5	1	1	1	1
20	0.1	0.1	0.1	0.3	1	1	1	1
22	0.1	0.1	0.1	0.1	0.9	0.9	0.9	0.9
24	0.1	0.1	0.1	0.1	0.9	0.9	0.9	0.9
26	0.1	0.1	0.1	0.1	0.7	0.9	0.9	0.9
28	0	0	0	0	0.5	0.9	0.9	0.9
30	0	0	0	0	0.4	0.9	0.9	0.9
32	0	0	0	0	0.2	0.9	0.9	0.9
34	0	0	0	0	0.1	0.8	0.9	0.9
36	0	0	0	0	0.1	0.8	0.9	0.9
38	0	0	0	0	0.1	0.7	0.8	0.8
40	0	0	0	0	0	0.6	0.8	0.8

Table 5: Ponderosa pine

DBH Category	Flamelength Category							
	2	4	6	8	10	12	14	16
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
4	0.7	1	1	1	1	1	1	1
6	0.7	0.7	1	1	1	1	1	1
8	0.6	0.6	0.9	1	1	1	1	1
10	0.6	0.6	0.7	1	1	1	1	1
12	0.5	0.5	0.5	1	1	1	1	1
14	0.5	0.5	0.5	0.9	1	1	1	1
16	0.4	0.4	0.4	0.7	1	1	1	1
18	0.4	0.4	0.4	0.5	1	1	1	1
20	0.3	0.3	0.3	0.3	0.9	1	1	1
22	0.3	0.3	0.3	0.3	0.8	1	1	1
24	0.3	0.3	0.3	0.3	0.6	1	1	1
26	0.2	0.2	0.2	0.2	0.4	0.9	1	1
28	0.2	0.2	0.2	0.2	0.3	0.9	1	1
30	0.2	0.2	0.2	0.2	0.2	0.8	1	1
32	0.2	0.2	0.2	0.2	0.2	0.7	1	1
34	0.1	0.1	0.1	0.1	0.1	0.5	0.9	1
36	0.1	0.1	0.1	0.1	0.1	0.4	0.9	1
38	0.1	0.1	0.1	0.1	0.1	0.3	0.8	1
40	0.1	0.1	0.1	0.1	0.1	0.2	0.8	1

Table 6: Sugar pine

DBH Category	Flamelength Category							
	2	4	6	8	10	12	14	16
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
4	0.7	1	1	1	1	1	1	1
6	0.6	0.9	1	1	1	1	1	1
8	0.5	0.6	1	1	1	1	1	1
10	0.4	0.4	0.9	1	1	1	1	1
12	0.4	0.4	0.7	1	1	1	1	1
14	0.3	0.3	0.5	0.9	1	1	1	1
16	0.2	0.2	0.3	0.8	1	1	1	1
18	0.2	0.2	0.2	0.7	1	1	1	1
20	0.2	0.2	0.2	0.5	0.9	1	1	1
22	0.1	0.1	0.1	0.4	0.8	1	1	1
24	0.1	0.1	0.1	0.3	0.7	0.9	1	1
26	0.1	0.1	0.1	0.2	0.6	0.9	1	1
28	0.1	0.1	0.1	0.1	0.5	0.9	0.9	0.9
30	0.1	0.1	0.1	0.1	0.4	0.8	0.9	0.9
32	0.1	0.1	0.1	0.1	0.3	0.7	0.9	0.9
34	0.1	0.1	0.1	0.1	0.2	0.6	0.9	0.9
36	0	0	0	0.1	0.2	0.5	0.8	0.9
38	0	0	0	0	0.1	0.4	0.8	0.9
40	0	0	0	0	0.1	0.3	0.7	0.9

Table 7: White fir

Appendix E: Hazard Analysis

There are two separate hazard analyses which are then combined to form a single overall episodic disturbance hazard analysis for the watershed. Both occur at specific times during stage three of the modeling process.

The insect hazard analysis is a simple look at the landscape to determine if individual stands would be affected by insects if the current simulation period was a drought type period. Regardless of what the weather pattern actually is for the current simulation period the insect hazard analysis assumes there is drought type weather; it's a "what if" scenario. It is a Yes-No analysis and no attempt is made to calculate actual severity. The rules for determining whether or not a stand is hit by insects are found in the section titled Apply Insect Disturbance. This analysis is done on a stand-by-stand basis. Once completed the SafeD model stores a 0-1 variable indicating whether or not each particular stand is susceptible to insect mortality.

The flame hazard analysis is a more complicated look at the landscape involving the calculation of a flame height for each stand under the assumption that a theoretical fire has occurred in the stand. An external program called FLAMMAP is used for this analysis. FLAMMAP is a raster-based program that takes particular landscape attributes and weather stream information to calculate a flame height for each pixel; in our case each pixel is a stand. FLAMMAP is a recently developed program written by the same developer of FARSITE (Finney, 1999). Our use of FLAMMAP is on an experimental basis. The science team felt comfortable using FLAMMAP at this time because of related work completed by Finney and the Fire Science Laboratory (see previous Apply Fire Disturbance section). Future work on the SafeD model should incorporate better documentation on FLAMMAP as it becomes available.

What is important to understand about our flame hazard analysis (in relation to my thesis) is how the SafeD model prepares the input data for FLAMMAP. FLAMMAP needs five landscape attribute grids and several parameterization and weather stream files to work. The five landscape attribute grids are: fuel model, canopy closure, height to live crown, crown bulk density, and stand height. This information is currently stored within the SafeD model and it must be exported to an external ASCII file that the

FLAMMAP program can read. The parameterization files are some "setup" files that FLAMMAP looks for to calibrate itself. These files include information about where files are to be stored and read; what type of output files are wanted from FLAMMAP; and where the weather stream files are located.

In any given simulation period the flame hazard analysis is started by having the SafeD model export the current five landscape attribute grids to the hard drive of the computer and create the needed parameterization files for FLAMMAP. The SafeD model then "calls up" FLAMMAP. Upon completion FLAMMAP will create a landscape grid that represents the potential flame height for each cell. The SafeD model reads that information in and temporarily stores it. Flame height is an important indicator of the potential effects a fire will have on a stand. Flame height can be related through the First Order Fire Effects Model (FOFEM) to obtain tree mortality (as discussed in the Apply Fire Disturbance section). The flame hazard analysis is completed by dividing the calculated flame height into four hazard categories: Low, 0-4 feet; Moderate, 4-8 feet; High, 8-12 feet; Extreme, > 12 feet.

The last step of the overall hazard analysis is the combining of the insect and fire hazard analyses. This actually occurs outside of the SafeD model. As I described above, both the insect and fire hazard analyses output data to the hard drive during each simulation period. At the end of the entire simulation I combine the two analyses from each period into a single insect-fire hazard rating through a GIS. It should be noted that we are currently using these analyses for mapping purposes only. This is why the combining process is done after the entire simulation and outside of the SafeD model - it saves computing time.

Appendix F: The SafeD Model Code

MAIN.CPP

```
/*****************************************************************************

  Start of the BIG program to run SafeD, which will:
  1) Call up Premo(heidi's prescription generator) and get prescriptions
  2) Pick and optimize prescription selection
  3) Initiate episodic disturbances
  4) React to disturbances
  5) [eventually: Re-optimize stand prescriptions and prescription selection]

-- Coding started 20 Nov, 1998                  David Graetz
*****************************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"                    //to hold global DEFINES, etc..
#include "main.h"                       //external functions called up within main()
#include "data.h"                       //Hold the prototypes for various structures
//*************************************************************************************************

ulong NATLN;                  // A global variable to hold the NextAvaiableTreeListNumber
int WEATHER;                  // To pass along what the weather status is for a period
int SimWeather[NP]; // To hold the weather status for entire simulation

//A global array to use for toggling whether to run certain portions at different period (1=yes, 0=no) Can only
have 4 periods with 1
int EvaluateThisPeriod[NP];                                          //see EvaluatePeriods() in misc.cpp

int UniqueMinor[MAX_SUBWATERSHEDS];              //To hold the ID's for all the different sub-watersheds
int USW;                                                                  //The number of Unique Sub-
Watersheds
// *****************************************************************************
// *****                                                    main
                                                    *******
// *****                                                  SafeD PROGRAM
        *******
// *****
                                                                  *******
// *****************************************************************************
// *****************************************************************************
int main()
{

// A couple of variables to hold temporary names of files
char run_flammap[250]="", run_farsite[250]="";
ulong FirstTreelistThisPeriod;

//------------------------- End of variable defining ------------------------------
//Seed the random number generator
srand(time(NULL));

//=============================================================
//Ensure that all the needed directories are made
#ifdef MAKE_DIRECTORIES
MakeDirectories();
exit(0);
#endif
//=============================================================

EvaluatePeriods(EvaluateThisPeriod);             //Fill up the EvaluateThisPeriod array

//Set the grid environment scope for amls - This must be done after EvaluatePeriods()!!
EnvScope(EvaluateThisPeriod);

WeatherStatus(SimWeather);                                          //Fill up the SimWeather array with weather
type
                                                                                                    //1=
Wet, 2=Moderate, 3= Drought, 4 = Severe Drought
DeleteOldStuff();                                                          //Clean up the
\outputs\prescription\modeled\* directory before each run

StartTreeDamageFile();                                                     //Create a new file that can be
opened later in program in append mode

#ifdef FRAMEWORK_PROJECT
CopyExecutables();
#endif

#ifdef NEW_LANDSCAPE_FILES
InitialFiles();                                                                    //Get the initial GIS
files ready for per0
exit(1);                      //for testing
#endif                                                                                    //Be sure to
set the correct global environment #define

#ifdef FREQUENCY_ANALYSIS                                           //toggle which ones and always exit after
TimingChoiceFrequency();
exit(1);
#endif

//=============================================================================================================
====
//*** ALWAYS NEED !!!! ****
```

```
CreateMainData();                                                    //Make the initial Data.* structure
arrays
                                    //REMEMBER: The Data.Goal and Data.Hold arrays get filled here if #define
RERUN_SIM, otherwise
                                    // they will be picked and inputted into the Data.* arrays in
PickPrescriptions()
//=============================================================================================================
=====

// ****************************************************************
//*** ALWAYS NEED !!!! ****
//Fill up the global UniqueMinor[] array and get the global USW value
USW = CountSubWatersheds(UniqueMinor) ;
// ****************************************************************


//=============================================================================================
#ifdef OWNERSHIP_ANALYSIS                                            //must be after CreateMainData - needs info
from that
OwnershipByMinor(USW, UniqueMinor);
exit(1);
#endif
//=============================================================================================


// ********************************************************************************************************
//WARNING - Use the below function ONLY when I need to COMPLETELY redo initial prescriptions (see globals.h)
#ifdef INITIAL_PREMO
//InitialPremo();                                                    //Run StandOpt.exe for all the
goals, for all the initial stands
CreateSortedPremoBinaryFile();
exit(0);  //for testing
#endif
// ********************************************************************************************************

//=============================================
//*** ALWAYS NEED !!!! ****
//Get the initial fuel model and fuel loadings
InitialFuelController();

//Also initialize background ERA values for everyone
InitialEraValues();
//=============================================

// ********************************************************************************************************
//Determine or reload the prescriptions to use
#ifdef OPTIMIZE                                                      //and I don't want to use default Goal=9,
Hold=0
PickPrescriptions(GOAL_TO_USE);                  //see below for goal # meanings
/*
1 = Even-flow of timber by sub-watershed w/ full range of activities allowed in LSR's and Riparian reserves
2 = Grow Only
3 = The Finney-Effect "bricks" - for FRAMEWORK only
4 = Rx6 - of the Framework alternatives
*/
#else
ReuseBestPrescription(GOAL_TO_USE);
#endif
//exit(0);                    //for testing - if goal maps and stuff are needed be sure to run
OutputPreSimAnalysisData()
// ********************************************************************************************************


//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++
#ifdef ALL_GOALS_BIGTREES
//Do an analysis and see what the Big Trees values are if using one solid goal assignment across landscape (for all
10 goals).
//NOTE: needs to have the "InitOpt.bin" file create for the landscape goal, so PickPrescription() needs to have
been run for goal
OutputPotentialBigTreesAllStandGoals();
#endif
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++


//=============================================
//*** ALWAYS NEED !!!! ****
// Fill up the Data.*[] arrays with Premo data
FillInitialPremoData(0);                              //period 0 indicates an initial run,

//exit(1);
//=============================================

DoubleCheckVegcodes();

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//        Redo the HLC and CBD measurements
RedoHlcCbd();
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// ********************************************************************************************************
//NEED FOR ANALYSIS AND MAPS TO SHOW WHAT IS HAPPENING or HAPPENED BEFORE ENTERING SIMULATION PERIODS
#ifndef RERUN_SIM
if( OutputPreSimAnalysisData() == FALSE)
```

```
        Bailout(52);
#endif
//exit(0);                    //for testing
// ************************************************************************************************************

//goto END;

//Set the variable NATLN (Next Available Tree List Number) at this point and updated throughout program
NATLN = FIRST_AVAILABLE_TREELIST;

// ------------------------------------------------------------------------------------------------------------
------
// ---------------   All the above needs to take place before Period 1 occurs.   ---------------------------------
------
// ------------------------------------------------------------------------------------------------------------
------
for(period=1;period<=NP;period++)
{

        //*****************************************************************************************
        //      It is assumed, that when SafeD enters Period 1, that there has been 5 yrs of growth
        //      and harvest in Premo.  Premo works by taking a Time 0 treelist, harvesting (if at all) right
        //      away and then growing forward 5 years (and accounting for periodic mortality).  So SafeD will
        //      assume that the Premo values stored for Period 1 are values that represent activity and growth.
        //*****************************************************************************************

        //print some stuff to screen
        StartPeriodInfo(period);

        //Get the weather to pass on
        WEATHER = SimWeather[period-1];

//set weather to 3 while developing
//WEATHER = 3;

        //Delete the stuff(if any) in the Modified directory before ANY disturbance
        DeleteModified();

        //Adjust the fuel loadings for decay and new fuel loading contributions - and then calculate new fuel
model
        AdjustFuelStuffForGrowth(period);

        //============================          Start Episodic Disturbances
==================================================
        //                  At period 1, this means there has been 5yrs (1period) of harvest and growth (i.e.
management) before 1st disturbance
        //=============================================================================================================
====================

        FirstTreelistThisPeriod = NATLN;

        //Send out a file to show where the potential Bug problems are this period (assuming a drought year)
        MapPotentialBugs(period);


        // ****************** First disturbance to occur will be insects **********************
#ifdef USE_BUGS
        if( ApplyInsectDisturbance(period, WEATHER, FirstTreelistThisPeriod) == FALSE )
                Bailout(76);
#endif

        /*****************************************************************************************
                NOTE:  The insects did some damage and their effects to HLC, STANDHEIGHT, CBD, and CLOSURE
                were recalculated - but we do not have anything to update their effects to the fuel loadings
                thus the FuelModels are not changing. - This is OK according to Bernie 16Feb00
        *****************************************************************************************/

        //Output landscape data for Flammap and Farsite for this period (FUEL, HEIGHT, BLC, CBD, and CLOSURE)
        if( OutputCurrentLandscapeData(period) == FALSE)
                Bailout(50);

        //Output some random fuel loads and fuel model for evaluation
        OutputFuelLoadsModel(period);

        //********************** Prepare and run FLAMMAP ***********************************************
        if( PrepareFlammap(period,WEATHER) == FALSE)
                Bailout(0);

#ifdef USE_FLAMMAP          //******  RUN FLAMMAP and output the results to ...\outputs\\per*\flammap.asc
        if(EvaluateThisPeriod[period-1] == TRUE)
        {
                sprintf(run_flammap, "%s%s%d\\per%d\\runflammap.bat",PREFIX,INPUTS,GOAL_TO_USE,period);
                system(run_flammap);
                CleanAndSave(period,FLAMMAP,ACTUAL);
                InOutFlammapResults(period, ACTUAL);
        }
#endif

        //************************ Prepare and run FARSITE ***********************************************
        if( PrepareFarsite(period,WEATHER) == FALSE )
                Bailout(1);

#ifdef USE_FARSITE          //******  RUN FARSITE - apply the effects and update data
        sprintf(run_farsite, "%s%s%d\\per%d\\runfarsite.bat",PREFIX,INPUTS,GOAL_TO_USE,period);
        system(run_farsite);
```

```
          CleanAndSave(period,FARSITE,ACTUAL);
          ApplyFireDisturbance(period, FirstTreelistThisPeriod);              //stuff in CountFireHit() commented
out - double check before running for real
#endif    //use_farsite


          //After all the disturbances, figure out which stands were hit - make new Premo runs - and input the new
data
          if( period != NP )
                    //Don't do on last period!
                    ManageNewPremoRuns(FirstTreelistThisPeriod, period);

          //A double check to make sure vegcodes are OK before outputting
          DoubleCheckVegcodes();

          //Output the current VEGCODES for GIS mapping
          if(EvaluateThisPeriod[period-1] == TRUE)
                    OutputVegcodes(period);

          EndPeriodInfo(period);
}//end of for(period=1;period<=NP;period++)

// ------------------------------------------------------------------------------------------------------------------
------
// ------------------------------------------ END OF PERIOD RUNS ----------------------------------------------------
------
// ------------------------------------------------------------------------------------------------------------------
------
//END:

//ANALYSIS AND MAPS TO SHOW WHAT HAPPENED AFTER THE SIMULATION
if( OutputPostSimAnalysisData() == FALSE)
          Bailout(52);


return TRUE;
}//end main program


MAIN.H

//------------ Variable and functions called or used in main() ------------------------


//variables to use within main()
int period;

//define in StandOptStuff.cpp
extern void InitialPremo(void);
extern void FillInitialPremoData(int per);
extern void CreateSortedPremoBinaryFile(void);
extern int ManageNewPremoRuns(ulong FTL, int Per);


//defined in ReadData.cpp
extern int CreateMainData(void);

//defined in PrepareFarsite.cpp
extern int PrepareFarsite(int period, int weather);
extern void InitialFiles(void);

//defined in PrepareFlammap.cpp
extern int PrepareFlammap(int period, int weather);
extern void InOutFlammapResults(int p, int Status);

//defined in ArcInfoController.cpp                -- These are not called up anymore
extern void VectorResults(int p);
extern void VegCodeMapping(int status);

//defined in FireEffects.cpp
extern int ApplyFireDisturbance(int period,  ulong FTTP);
extern void DoubleCheckVegcodes(void);


//defined in Misc.cpp
extern void MakeDirectories(void);
extern void EnvScope(int Eval[NP]);
extern int CountSubWatersheds(int UM[]);
extern void StartPeriodInfo(int p);
extern void         EndPeriodInfo(int p);
extern void DeleteOldStuff(void);
extern int WeatherStatus(int Weather[NP]);
extern int EvaluatePeriods(int Eval[NP]);
extern void Bailout(int ErrorNumber);
extern void CleanAndSave(int Per, int Program, int Status);
extern void DeleteModified(void);
extern void CopyExecutables(void);
extern void StartTreeDamageFile(void);


//defined in goal_controller.cpp
extern void PickPrescriptions(int goal);
extern void ReuseBestPrescription(int goal);

//defined in OutputData.cpp
```

```
extern int OutputCurrentLandscapeData(int Per);
extern int OutputPreSimAnalysisData(void);
extern int OutputPostSimAnalysisData(void);
extern void TimingChoiceFrequency(void);
extern void OwnershipByMinor(int USW, int UniqueMinor[]);
extern void OutputFuelLoadsModel(int Per);
extern void OutputPotentialBigTreesAllStandGoals(void);
extern void OutputVegcodes(int Per);

//defined in Insects.cpp
extern int ApplyInsectDisturbance(int Per, int Weather, ulong FTTP);
extern void MapPotentialBugs(int Per);


//defined in CommonDisturbance.cpp
extern void StartPtrTpInfo(struct PTR_TFP *ptr_info, ulong FirstTreelist);

//define in FuelEra.cpp
extern void InitialFuelController(void);
extern void AdjustFuelStuffForGrowth(int ActualPer);


//define in EraStuff.cpp
extern void InitialEraValues(void);

//defined in StandData.cpp
extern void RedoHlcCbd(void);
```

GLOBALS.H

```
// ***************************************************************************************************************
// -----------------     Common Macros and #define for any project                    ----------------------
// ***************************************************************************************************************

//Make sure only one "*_PROJECT" is used below - will tell which *_Globals.h file to use
#define APPLEGATE_PROJECT
//#define FRAMEWORK_PROJECT


//=============================================================================================================
//=============================================================================================================

//Miscellaneous DO NOT CHANGE
typedef unsigned short ushort;
typedef unsigned long ulong;
#define FALSE                                   0
#define TRUE                                    1
#define REAL                                    0
#define FAKE                                    1
#define PREDICTED                       0                               //used as a toggle when inputting some data
(e.g. InputFlammap() )
#define ACTUAL                                  1                               //
                                        "
#define REUSE                                   2
#define LAST                                    3
#define FLAMMAP                                 1                               //  toggle to indicate stuff for
FLAMMAP
#define FARSITE                                 2                               //  toggle to indicate stuff for
FARSITE
#define SAFED                                   3                               //  for general SAFED stuff
#define SWAP1                                   1                               //  toggle for a one-swap move
#define SWAP2                                   2                               //  toggle for a two-swap move
#define FILE_TYPE                       2                               // 1 = AsciiFiles,    2 = BINARY files


//*****  TOGGLE switches to indicate whether or not to use certain parts of code (comment or uncomment as needed)
//#define NEW_LANDSCAPE_FILES           //when changes have been made to original GIS data - will call up ArcInfo
and create new
//#define MAKE_DIRECTORIES                      //for a new project using a new DRIVE PREFIX only
//#define SAVE_FOR_RERUN                                //use to save certain large files if RERUN_SIM is going
to be used
#define RERUN_SIM                               //use if this is simulation run is Identical to
previous run and want comparison
//#define OPTIMIZE                              //Otherwise, Goal defaults to 9 and Hold defaults to 0
//#define INITIAL_PREMO                         //Run PREMO for all the initial stands

// *** These toggles should almost always be on when running full, actual simulations
#define CREATE_TREE_INDEX                       //Toggle for the CreateTreeIndex function call in ReadData.cpp
#define USE_BUGS
#define USE_FLAMMAP                             //Whether or not to actually run FLAMMAP each period
#define USE_FARSITE                             //Whether or not to actually run FARSITE each period
#define END_PERIOD_PREMO                        //Whether or not to actually run PREMO at end of period

//*****  ALWAYS USE THE BEST GOAL-HOLD FOUND IF NOT ACTUALLY OPTIMIZING ON THIS RUN
#ifndef OPTIMIZE
#define USE_BEST_GOAL_HOLD                      //Only to read in a previous goal and hold solution found (used
in ReadData.cpp)
#endif

//***** Switches to use for PRE or POST-SIMULATION ANALYSIS - will control what data gets outputted - may not need
all these
#define ACRES_HARVEST
#define MAP_GOALS

//*****  DIFFERENT ANALYSIS TO RUN BEFORE SIMULATIONS - called up in main.cpp and have an exit(1) statement after
```

```
//#define FREQUENCY_ANALYSIS                    //to evaluate harvest timing choices from prescriptions
//#define OWNERSHIP_ANALYSIS                    //to see what the % ownership is by 6th field subwatershed
#define ALL_GOALS_BIGTREES                              //to see BigTrees for one landscape goal for each of
the stand goals applied only

//*****  Goal variables
#define APPLE_ERA             1
#define GROW_ONLY             2
#define FINNEY_EFFECT         3                                                         //The finney "bricks"
#define RX6                                   4
        //JoAnn's rules to mimic alternative 6 of the Framework stuff
#define GOAL_TO_USE           (GROW_ONLY)             //which of the landscape goals to run - see main.cpp
for values

//***** Set a little error checker for conflicts with the above #defines - will print up on debug window in
compiler
#if defined(RERUN_SIM) && defined(OPTIMIZE)
#error ERROR: Can't have RERUN_SIM and SAVE_GOAL_HOLD defined at the same time
#endif

#if defined(RERUN_SIM) && defined(SAVE_FOR_RERUN)
#error ERROR: Can't have RERUN_SIM and SAVE_FOR_RERUN defined at the same time
#endif

#if defined(RERUN_SIM) && defined(PREDICTED_FLAMMAP)
#error ERROR: Can't have RERUN_SIM and PREDICTED_FLAMMAP defined at the same time
#endif

#if defined(APPLEGATE_PROJECT) && GOAL_TO_USE == FINNEY_EFFECT
#error ERROR: The Applegate project does not have rules set up to apply the Finney Effect
#endif

#if defined(APPLEGATE_PROJECT) && GOAL_TO_USE == RX6
#error ERROR: The Applegate project does not have rules set up to apply the RX6 landscape goal
#endif


//Stand Goal globals
#define SG_FIRE                               0
#define SG_INSECTS                    1
#define SG_FISH                               2
#define SG_WILDLIFE_S                 3
#define SG_WILDLIFE_C                 4
#define SG_PNV                                5
#define SG_COMBO1                     6
#define SG_COMBO2                     7
#define SG_COMBO_ALL                  8
#define SG_GROWONLY                   9



//***** Switched for which heuristic I want to use
#define DELUGE
//#define ANNEAL

//***** Another prefix for filenames
#ifdef TABUSEARCH
#define OPTPREFIX "T"                                                   //Tag on to TabuSearch files
#elif defined(DELUGE)
#define OPTPREFIX "D"                                                   //Tag on to Deluge files
#else
#define OPTPREFIX "A"                                                   //Tag on to Simulated Annealing files
#endif

//***** for any heuristic
#define FIRST_SWAP1CHANCE                                               .50
                //50% chance of making a one-swap move during second 1/3 of DelugeLoops
#define SECOND_SWAP1CHANCE                                              .25
                //25% chance of making a one-swap move
#define PRINT_LOOPS                                                             2000


#if(GOAL_TO_USE == 1)
#define BASE_ADJ                                                                (double)1
        //Adjustment to the Baseline, determined in GetBaselineVTO()
#else
#define BASE_ADJ                                                                (double)1.5

#endif

//***** Some stuff for TABU search
#define PENALTY1              1000000                   //penalty assigned when current "move" is already in
solution or is TABU
#define TABULOOP              100                                       //Number of iterations to run the tabu search
#define TLONG                                 15                                //Number of times a move can enter
TabuLong before getting a penalty in Short
#define TSHORT                                11                                //Number of iterations a move can't
come back into solution in the short term
#define LT_PENALTY            6                                         //Penalty assigned to Short because move has
enter TLONG times

//***** Stuff for The Great Deluge
#if(GOAL_TO_USE == 1)
#define LOOP_FACTOR                                                     13
                //Multiplied by # of cells in solution-control how many deluge loops
```

```
#define RAIN                                                                    (double).001
                //The RAIN amount to use
#else
#define LOOP_FACTOR                                                 10

#define RAIN                                                                    (double).01

#endif


//***** Stuff for Simulating Annealing
#if(GOAL_TO_USE == 1)
#define INITIAL_TEMP                                        1000000000
#define LOOPS_AT_ONE_TEMP                       200
#define COOLING_RATE                                        .99
#define MIN_TEMP                                            1000
#define DELTA_FACTOR                                        4
#define PENALTY                                                     10
#else           //default
#define INITIAL_TEMP                                        50000000000
#define LOOPS_AT_ONE_TEMP                       250
#define COOLING_RATE                                        .985
#define MIN_TEMP                                            500000
#define DELTA_FACTOR                                        3
#define PENALTY                                                     10
#endif


//***** Common Math values & EXPansion values
#define DG2RD       0.017453292                                             // PI / 180
        Degree to Radians
#define RD2DG       57.29577951                                             // 180 / PI
        Radians to Degrees
#define M2FT        3.28084                                                 // Meters to feet
#define FT2M        .3048                                                   // Feet to Meters
#define PI                                                  3.141592653589
#define BASAL_CONSTANT                  (PI / (4*144) )     //Formula for BA is:  (PI * D-squared) / (4 * 144)
                - this replaces all but D-squared
#define TONS                                    2000                        //lbs per ton
#define FUEL_LOAD_EXP                   10                                  //value to * real fuel loadings to
fit ushort and not lose precision
#define   BASAL_EXP                     10                                  //Expansion value for Basal area to
fit ushort
#define   DENSITY_EXP                           100                                 //Expansion value for
CBDensity to fit ushort
#define   BIGTREES_EXP                  10                                  //Expansion value for BigTrees to
fit ushort
#define   ERA_EXP                               100                                 //Expansion value for Era
to fit ushort
#define BIG_TREE_SIZE                    30                                 //In INCES, size of big trees we
are evaluating

//*****************************************************************************
//------------- The directory paths that are used throughout the SafeD program ----------
//*****************************************************************************
#define INPUTS                                              "\\model\\inputs\\goal"
#define ConstantInput                           "\\model\\inputs\\Constant"
#define CommonInitial                           "\\model\\inputs\\CommonInitial"
#define TREE_INDEX                              "treeindex.txt"
#define IT_INDEX                                "InitialTreeindex.txt"


#define PremoProgName                           "\\model\\standopt\\Premo\\Debug\\Premo.exe"
#define FarsiteName                             "\\model\\farsite\\farsite\\Debug\\farsite.exe"
#define FlammapName                             "\\model\\flammap\\flammap\\Debug\\flammap.exe"
#define FarsiteOutput1                          "\\model\\SafeD\\per"
#define RerunDir                                "\\model\\RerunData\\goal"
#define AmlDir                                              "\\model\\amls"

#define OUTPUTS                                             "\\model\\outputs\\goal"
#define VectorOutDir                            "\\model\\outputs\\vector_out"
#define RasterOutDir                            "\\model\\outputs\\raster_out"
#define ErrorDir                                "\\model\\outputs\\Errors"
#define MapDir                                              "\\model\\outputs\\final_maps"
#define GeneralDataDir                          "\\model\\outputs\\GeneralData"
#define PreSimOutputDir                         "\\model\\outputs\\PreSimData\\goal"
#define PostSimOutputDir                "\\model\\outputs\\PostSimData\\goal"
#define OutputDelugeDir                                     "\\model\\outputs\\Deluge\\goal"
#define InitialStandDataDir             "\\model\\outputs\\StandData\\Initial"
#define ModeledStandDataDir             "\\model\\outputs\\StandData\\Modeled"
#define InitialPresDir                          "\\model\\outputs\\prescriptions\\initial"
#define ModeledPresDir                          "\\model\\outputs\\prescriptions\\modeled"
#define P_ToModDir                              "\\model\\outputs\\prescriptions\\ToModify"
#define P_ModDir                                "\\model\\outputs\\prescriptions\\Modified"


//*****************************************************************************
//------------- Some DEBUG toggles that can be used to turn on/off printf statements -----
//*****************************************************************************
#define DEBUG_COUNTCELLID
#define DEBUG_VEGCODES
#define DEBUG_MAINDATA
#define DEBUG_IGPOINTS
#define DEBUG_FLAMLAYERS
#define DEBUG_FLAMMAPENVT
```

```
#define DEBUG_COUNTSUB
#define DEBUG_DELUGE
#define DEBUG_OBJVALUES
#define DEBUG_DECREASESHORT
#define DEBUG_DELUGEGOAL1
#define DEBUG_MAINGOAL2
#define DEBUG_INITIAL_GOAL2
#define DEBUG_FILLVALUESTOOPTIMIZE
#define DEBUG_LOA


//*************************************************************************************************************
//        This must be last because many of the macros in the below file require #defines from above
//*************************************************************************************************************
#ifdef APPLEGATE_PROJECT
#include "Applegate_Globals.h"
#elif defined(FRAMEWORK_PROJECT)
#include "Framework_Globals.h"
#endif


APPLEGATE_GLOBALS.H

// *************************************************************************************************************
// -----------------     Specific macros and #defines for the APPLEGATE project       ----------------------
// *************************************************************************************************************
#define MAIN_USER "applegate"                                   //pass this to any AMLs so they can set proper
variables
#define SHORT_NAME "apple"                                              //mostly used in PrepareFarsite() and
PrepareFlammap - for misc. files used by those programs
#define PREFIX "g:"                                             //Used to set the beginning directory path
for sprintf() calls
#define PREMO_TOGGLE 1

// ---------------------- Uncomment the environment to use ------------------------------------------
#define WHOLE_RUN
//#define LITTLE_RUN
//#define COMPARE_RUN
//#define TINY_RUN
// --------------------------------

//For output .asc files to use in ArcInfo
#define CELLSIZE  25                                                                            //MUST BE IN
METERS
#define NODATA      -9999

//cell-dependent math conversions
#define ACREEQ      (CELLSIZE * CELLSIZE * .000247)             //The equivalent acres in one cell


#define NODATAFLAG                      65000           //indicates input data theme had a NODATA value in this
cell, but there
                                                                                                        //was a valid
Cellid. 65000 works as long as no theme data is suppose
                                                                                                        //to have that
value.
#define NONFOREST                       209                             //Easier way to track - this is the flag for
non-forest types
                                                                                                        //Also, because
my treelist values may be >= than NODATAFLAG (65000)
                                                                                                        //give it the
209 flag which already gets nothing done to it in PREMO
#define FIRST_AVAILABLE_TREELIST        210
#define FUEL_FLAG                                       999

//For loops, etc
#define NP                                              8                       //Number of Periods
#define YIP                                             5                       //Years In Period
#define GOALS                                   10              //# of stand goals (in PREMO) that we can
have
#define HOLDNO                                  2               //How many "HoldFor" periods we have to
evaluate from prescription generator
#define LANDSCAPE_GOALS                 2               //The number of landscape goals programmed in - used in
making directories for new projects
#define MAX_SUBWATERSHEDS     221               //The max number of Unique 6th-field Subwatersheds (use Max value
+ 1 to handle NODATA in GIS)
#define WATER_BODY                      220             //ID assigned to water bodies in the subwatershed GIS
coverage
#define VEGCLASSES                      13              //The number of original Vegetation Classes from
initial classification
#define STAGES                                  15              //The number of original Seral Stage Classes
from initial classification
#define FILES                                   12              //The number of input landscape files - leave
at 12 (don't need PRULE used in Framework stuff)

//Misc variables to use with the original GIS layers
#define IN_BUFFER   100                                 //Whether or not a cell is in the Fed stream buffers
(Fed lands only) for Data.Buffer[]
#define IN_OLDFIRE  100                                 //Whether or not a cell is in an old fire polygon, for
Data.FireHistory[]

//***** Variable for the Subwatershed ERA threshold - can vary for each period
#define INITIAL_TRYS 5                                          //How many times to use the below ERA
threshold values before failing
```

```
#define PER1_ERA    12                          // is really 0.28 or something like that (i.e. divided by ERA_EXP
)
#define PER2_ERA    12
#define PER3_ERA    12
#define PER4_ERA    12
#define PER5_ERA    12
#define PER6_ERA    10
#define PER7_ERA    9
#define PER8_ERA    9                           //There must be at least the same number as Number of Periods

//Codes to use when calling up FillPremoData after a disturbance - have no meaning, only for checking
#define FIRE        9997
#define BUGS        9998

//Codes to use when evaluating OWNERSHIP - these codes were used in Finalown.own_code of GIS data
#define OWN_PNI             33                          //Private Non-Industrial
#define OWN_PI              36                          //Private Industrial
#define OWN_BLM             69                          //Bureau of Land Management
#define OWN_USFS    76                          //US Forest Service
#define OWN_STATE   89                          //State Lands
#define OWN_MISC    124                         //Miscellaneous owner

//Codes to use when evaluating LAND ALLOCATION - the codes were used in Finalown.land_code of GIS data
#define ALLOC_NF            0                   //Non-Federal and Private lands
#define ALLOC_RESERVE       41                  //Federal "Late Successional Reserves"
#define ALLOC_WILD          51                  //Federal "Wilderness"
#define ALLOC_MATRIX        71                  //Federal "Matrix" lands

//GIS Codes for Plant Association Groups
#define PAG_DFDRY                   1                               //These are the values generated in GIS when
creating the Pag layer
#define PAG_DFWET                   2
#define PAG_JEFFPINE                3
#define PAG_REDFIR                  4
#define PAG_PINEOAK                 5
#define PAG_WFDRY                   6
#define PAG_WFWET                   7
#define PAG_WATER                   8
#define PAG_BARREN                  9

//GIS Codes for intial vegetation classification
#define GIS_BARREN          1
#define GIS_WATER           2
#define GIS_SHRUB           3
#define GIS_GRASS           4

//This is really for the Framework - but I need the PRULE to be defined within the code....ignore for the Applegate
#if (GOAL_TO_USE == FINNEY_EFFECT)
#define PRULE               2                                       //which "Prescription RULE" grid to use for
the run 1=JoAnn's,  2=FinneyEffect
#else
#define PRULE               1
#endif
//=====================================================
//        CODES SPECIFIC FOR ACTUAL TREELIST DATA
//=====================================================

#define TOTALSP                     10                              //The number of treelist species codes there
are
#define NO_TALL_TREES       5                           //The number of tallest trees to use as a group to
average out and get the stand.height

//Codes for species within Treelist
#define BLACKOAK            0
#define DOUGFIR                     1
#define ICEDAR                      2
#define KPINE                       3
#define MADRONE                     4
#define PPINE                       5
#define RFIR                        6
#define SPINE                       7
#define   TANOAK                    8
#define WFIR                        9

//"Status" codes within Treelist
#define SNAG                        0
#define LIVE                        1
#define DWD                                 2

//Premo codes for VegClassification             VC is "Veg Class"
#define VC_CH                               1
#define VC_DH                               2
#define VC_EH                               3
#define VC_KP                               4
#define VC_MC                               5
#define VC_OPEN                             6
#define VC_PINE                             7
#define VC_RF                               8
#define VC_WF                               9
#define VC_MC3                              10

//***** Variables for EvaluateThisPeriod....put them in chronological order
#define PER1        1                                               //actual year is (5yr * X ) ....i.e. 5 for
this one
#define PER2        4
```

```
#define PER3        6
#define PER4        8
#if( PER4 > NP)
#error ERROR: Can't have a higher evaluation period than the Number of Periods (NP) in the simulation
#endif

#ifdef WHOLE_RUN
#define ENVT "whole"
#define ROWS 2497
#define COLUMNS 3071
#define XLL 445525
#define YLL 4638800
#define UNIQUE 3191999
#define F_XLL 445525.000000
#define F_YLL 4638800.000000
#define MOC                                                (5280 * FT2M / CELLSIZE) //The number of cells it takes
to make a linear mile ( Mile Of Cells = MOC )
#endif

#ifdef LITTLE_RUN
#define ENVT "little"
#define ROWS 945
#define COLUMNS 976
#define XLL 496385
#define YLL 4653248
#define UNIQUE 467878
#define F_XLL 496385.906250
#define F_YLL 4653248.000000
#define MOC                                                30                    //see #define MOC in #ifdef
WHOLE_RUN- lowering because this test area is too small
#endif

#ifdef COMPARE_RUN
#define ENVT "compare"
#define ROWS 264
#define COLUMNS 363
#define XLL 479753
#define YLL 4646797
#define UNIQUE 95832
#define F_XLL 479753.851912
#define F_YLL 4646797.395176
#define MOC                                                5                     //see #define MOC in #ifdef
WHOLE_RUN- lowering because this test area is too small
#endif

#ifdef TINY_RUN
#define ENVT "tiny"
#define ROWS 12
#define COLUMNS 12
#define XLL 488517
#define YLL 4653108
#define UNIQUE 144
#define F_XLL 488517.851912
#define F_YLL 4653108.395176
#define MOC                                                1                     //see #define MOC in #ifdef
WHOLE_RUN- lowering because this test area is too small
#endif
```

MISC.CPP

```
/*  This Misc.cpp code will hold Miscellaneous functions used by various other
pieces of the SafeD program.  They are here because at the time of
construction I did not think they fit anywhere more specific
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <direct.h>
#include <time.h>
#include "globals.h"
#include "data.h"


//functions used in Misc.cpp
void MakeDirectories(void);
void Bailout(int ErrorNumber);
void EnvScope(int Eval[NP]);
int CountSubWatersheds(int UM[]);
void StartPeriodInfo(int p);
void EndPeriodInfo(int p);
void DeleteOldStuff(void);
void DeleteInitialStuff(void);
int WeatherStatus(int Weather[NP]);
int EvaluatePeriods(int Eval[NP]);
void CleanAndSave(int Per, int Program, int Status);
int FillSubEraValues(int SubEra[]);
void DeleteModified(void);
void DeleteToModify(void);
void CopyExecutables(void);
void PrintToStat(int Line, ulong Value);
```

```
void StartTreeDamageFile(void);

//Global to use for printing out total # of subwatershed
extern int USW;

//defined in FlammapStuff.cpp - used here again to make sure that file is deleted
extern void DeleteFar(int p);

//*************************************************************************************************
void PrintToStat(int Line, ulong Value)
//*************************************************************************************************
{
/*
This function will print various STATistic information to a file called ...\presimdata\goal3\Stats.txt
The file will be created on first call, otherwise appended to.  See switch statement below for
what gets printed out.
*/
FILE *OUT;
char Temp[300];
//------------------------ End of variable defining ----------------------------------------

//Always create the correct file name
sprintf(Temp, "%s%s%d\\stats.txt",PREFIX,PostSimOutputDir,GOAL_TO_USE);

//Open the file in the correct mode
if( Line == 1 )
        OUT = fopen(Temp, "w");
else
        OUT = fopen(Temp, "a+");



switch(Line)
{
case 1:
        fprintf(OUT,"TOTAL CELLS: %lu    \t\t%.2lf acres\n",Value,(double)Value * ACREEQ);
        break;
case 2:
        fprintf(OUT,"FORESTED CELLS: %lu    \t\t%.2lf acres\n",Value,(double)Value * ACREEQ);
        break;
case 3:
        fprintf(OUT,"CELLS IN SOLUTION: %lu    \t\t%.2lf acres\n",Value,(double)Value * ACREEQ);
        break;
case 4:
        fprintf(OUT,"TOTAL SUBWATERSHED:          \t\t\t%d\n",USW);
        fprintf(OUT,"SUBWATERSHEDS IN SOLUTION:   \t\t\t%lu\n",Value);
        break;
case 5:
        fprintf(OUT,"ACRES HIT BY FIRE:          \t\t\t%.2lf\n",(double)Value * ACREEQ);
        break;
case 6:
        fprintf(OUT,"ACRES HIT BY INSECTS:          \t\t\t%.2lf\n",(double)Value * ACREEQ);
        break;

default:
        fprintf(OUT, "No ideal what I'm printing - sending bad Line value to PrintToStat!!!\n");

}

fclose(OUT);

}//end PrintToStat


//*************************************************************************************************
void StartTreeDamageFile(void)
//*************************************************************************************************
{
/*
This function will create a new file for every simulation, which can get opened in Append+ mode during
each simulation and will contain data relating to how many/much trees are affected by either fire or
insects during given periods.
*/
FILE *OUT;
char Temp[300];
//------------------------ End of variable defining ----------------------------------------
//Always create the correct file name
sprintf(Temp, "%s%s\\goal%d\\TreeDamage.txt",PREFIX,GeneralDataDir,GOAL_TO_USE);

//Then just open and close the file to create it for later use - this will delete any old copies
OUT = fopen(Temp, "w");
fclose(OUT);

}//end StartTreeDamageFile

//*************************************************************************************
void Bailout(int ErrorNumber)
//*************************************************************************************
{
        //This function will create a text file with a error message before exiting
        //the program because of some error condition

char ErrorMessage[250];
```

```
FILE *WriteOut;
char filename[100];

        sprintf(filename, "%s%s\\Error.txt",PREFIX,ErrorDir);


        //Open up the Error.txt file
        WriteOut = fopen(filename,"w");                    //no error checking

//Get the appropriate ErrorMessage to write out
        switch(ErrorNumber)
        {
        case 0:   sprintf(ErrorMessage, "%s", "Something wrong in preparing FLAMMAP files");
                  printf("Something wrong in preparing FLAMMAP files\n");
                                                              break;
        case 1:   sprintf(ErrorMessage, "%s", "Something wrong in preparing FARSITE files");
                  printf("Something wrong in preparing FARSITE files\n");
                                                              break;
        case 2:   sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
CELLID.asc");
                  printf("Something wrong with the number of Rows and Columns in CELLID.asc\n");
                        break;
        case 3:   sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in CELLID.asc\n");
                  printf("Something wrong with the X and Y origins in CELLID.asc\n");
                                          break;
        case 4:   sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
TREELIST.asc");
                  printf("Something wrong with the number of Rows and Columns in TREELIST.asc\n");
                        break;
        case 5:   sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in TREELIST.asc\n");
                  printf("Something wrong with the X and Y origins in TREELIST.asc\n");
                                  break;
        case 6:   sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
OWNER.asc");
                  printf("Something wrong with the number of Rows and Columns in OWNER.asc\n");
                        break;
        case 7:   sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in OWNER.asc\n");
                  printf("Something wrong with the X and Y origins in OWNER.asc\n");
                                          break;
        case 8:   sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
ASPECT.asc");
                  printf("Something wrong with the number of Rows and Columns in ASPECT.asc\n");
                  break;
        case 9:   sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in ASPECT.asc\n");
                  printf("Something wrong with the X and Y origins in ASPECT25.asc\n");
                                  break;
        case 10: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in SLOPE.asc");
                  printf("Something wrong with the number of Rows and Columns in SLOPE.asc\n");
                        break;
        case 11: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in SLOPE.asc\n");
                  printf("Something wrong with the X and Y origins in SLOPE25.asc\n");
                                  break;
        case 12: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in ELEV.asc");
                  printf("Something wrong with the number of Rows and Columns in ELEV.asc\n");
                        break;
        case 13: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in ELEV.asc\n");
                  printf("Something wrong with the X and Y origins in ELEV.asc\n");
                                          break;
        case 14: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in MINOR.asc");
                  printf("Something wrong with the number of Rows and Columns in MINOR.asc\n");
                        break;
        case 15: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in MINOR.asc\n");
                  printf("Something wrong with the X and Y origins in MINOR.asc\n");
                                          break;
        case 16: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in ALLOC.asc");
                  printf("Something wrong with the number of Rows and Columns in ALLOC.asc\n");
                        break;
        case 17: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in ALLOC.asc\n");
                  printf("Something wrong with the X and Y origins in ALLOC.asc\n");
                                          break;
        case 18: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
STRBUF.asc");
                  printf("Something wrong with the number of Rows and Columns in STRBUF.asc\n");
                        break;
        case 19: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in STRBUF.asc\n");
                  printf("Something wrong with the X and Y origins in STRBUF.asc\n");
                                          break;
        case 20: sprintf(ErrorMessage, "%s", "There appears to be no treelist available for an IndexNo - see
screen");


                                  break;
        case 21: sprintf(ErrorMessage, "%s", "NOT READY FOR THAT TYPE OF DISTURBANCE in FillPremoData()\n");
                  printf("NOT READY FOR THAT TYPE OF DISTURBANCE in FillPremoData()\n");
                                          break;
        case 22: sprintf(ErrorMessage, "%s", "While updating Data.* arrays after disturbance (FillPremoData), New
Treelist # and Old Treelist # don't match - and they should!");
                  printf("While updating Data.* arrays after disturbance, New Treelist # and Old Treelist # don't
match - and they should!");


                                  break;
        case 23: sprintf(ErrorMessage, "%s", "Periods not matching while inputing V_*_*.txt (or SD*) file\n");
```

```
                printf("Periods not matching while inputing V_*_*.txt (or SD*) file\n");
                                        break;
        case 24: sprintf(ErrorMessage, "%s", "Could not create and fill the AllCFHarvest array in
Optimize.cpp\n");
                printf("Could not create and fill the AllCFHarvest array in Optimize.cpp\n");
                        break;
        case 25: sprintf(ErrorMessage, "%s", "Something wrong in generating an initial random solution for this
goal\n");
                printf("Something wrong in generating an initial random solution for this goal\n");


                                        break;
        case 26: sprintf(ErrorMessage, "%s", "Could not find an answer using TabuSearch for Goal
#%d",GOAL_TO_USE);
                printf("Could not find an answer using TabuSearch for Goal #%d",GOAL_TO_USE);
                                                                                        break;
        case 27: sprintf(ErrorMessage, "%s", "Could not input the solution for this goal - see screen");


                                        break;
        case 28: sprintf(ErrorMessage, "%s", "!!!!! WARNING !!!!! Not set up to handle more than %d HoldFor
values yet\n",HOLDNO);
                printf("!!!!! WARNING !!!!! Not set up to handle more than %d HoldFor values yet\n",HOLDNO);
        break;
        case 29: sprintf(ErrorMessage, "%s", "Something wrong in decreasing TabuShort");
                printf("Something wrong in decreasing TabuShort");
                                                                        break;
        case 30: sprintf(ErrorMessage, "%s", "Something wrong trying to print the Periodic Values from VTO");
                printf("Something wrong trying to print the Periodic Values from VTO");
                                        break;
        case 31: sprintf(ErrorMessage, "%s", "Mismatch of data while trying to Input a Solution");
                printf("Mismatch of data while trying to Input a Solution");
                                                break;
        case 32: sprintf(ErrorMessage, "%s", "While updating Data.Treelist (UpdateData() in
FireEffects.cpp)...Old and New Treelist #'s do not match...and they should!");
                printf("While updating Data.Treelist in FireEffects...Old and New Treelist #'s do not
match...and they should!");


                                        break;
        case 33: sprintf(ErrorMessage, "%s", "In FOPEM()...received a model species code > 9, which is invalid");
                printf("In FOPEM()...received a model species code > 9, which is invalid");
                                        break;
        case 34: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in VEG.asc");
                printf("Something wrong with the number of Rows and Columns in VEG.asc\n");
                                        break;
        case 35: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in VEG.asc\n");
                printf("Something wrong with the X and Y origins in VEG.asc\n");
                                        break;
        case 36: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in STAGE.asc");
                printf("Something wrong with the number of Rows and Columns in STAGE.asc\n");
                        break;
        case 37: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in STAGE.asc\n");
                printf("Something wrong with the X and Y origins in STAGE.asc\n");
                                        break;
        case 38: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
CLOSURE.asc");
                printf("Something wrong with the number of Rows and Columns in CLOSURE.asc\n");
                        break;
        case 39: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in CLOSURE.asc\n");
                printf("Something wrong with the X and Y origins in CLOSURE.asc\n");
                                        break;
        case 40: sprintf(ErrorMessage, "%s", "Looping is not set up in InitialStandOpt() to handle more than 2
HoldFors");
                printf("Looping is not set up in InitialStandOpt() to handle more than 2 HoldFors");
                break;
        case 41: sprintf(ErrorMessage, "%s", "HAVING TROUBLES FINIDING A TREELIST-GOAL-HOLD combo in the
ValueToOptimize array");
                printf("HAVING TROUBLES FINIDING A TREELIST-GOAL-HOLD combo in the ValueToOptimize array\n");
        break;
        case 42: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in FLAMMAP.fml
or FLAME.ASC");
                printf("Something wrong with the number of Rows and Columns in FLAMMAP.fml or FLAME.ASC\n");
                                        break;
        case 43: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in FLAMMAP.fml or
FLAME.ASC\n");
                printf("Something wrong with the X and Y origins in FLAMMAP.fml or FLAME.ASC\n");
                                                break;
        case 44: sprintf(ErrorMessage, "%s", "Could not find an answer using GreatDeluge or Annealing for Goal
#%d",GOAL_TO_USE);
                printf("Could not find an answer using GreatDeluge or Annealingfor Goal #%d",GOAL_TO_USE);
                break;
        case 45: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in GOAL.asc");
                printf("Something wrong with the number of Rows and Columns in GOAL.asc\n");
                        break;
        case 46: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in GOAL.asc\n");
                printf("Something wrong with the X and Y origins in GOAL.asc\n");
                                        break;
        case 47: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in HOLD.asc");
                printf("Something wrong with the number of Rows and Columns in HOLD.asc\n");
                        break;
        case 48: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in HOLD.asc\n");
                printf("Something wrong with the X and Y origins in HOLD.asc\n");
                                        break;
```

```
        case 49: sprintf(ErrorMessage, "%s", "Have a BASAL, CLOSURE, or CBD input value over 655 in a SD_*_*_.txt
file\n");
                 printf("Have a BASAL, CLOSURE, or CBD input value over 655 in a SD_*_*_.txt file\n");
                 break;
        case 50: sprintf(ErrorMessage, "%s", "Something wrong while trying to create new Flammap and Farsite
input data\n");
                 printf("Something wrong while making new Flammap and Farsite input data for next period\n");
        break;
        case 51: sprintf(ErrorMessage, "%s", "In Goal*.cpp, EligibleCell and AllocOK do NOT match - something
wrong\n");
                 printf("In Goal*.cpp, EligibleCell and AllocOK do NOT match - something wrong\n");
                          break;
        case 52: sprintf(ErrorMessage, "%s", "Something wrong while trying to output Analysis data\n");
                 printf("Something wrong while trying to output Analysis data\n");
                                          break;
        case 53: sprintf(ErrorMessage, "%s", "Can't harvest where there is a treelist of NONFOREST!!!\n");
                 printf("Can't harvest where there is a treelist of NONFOREST!!!\n");
                          break;
        case 54: sprintf(ErrorMessage, "%s", "Data.Vegcode has a code for water and the fuel model is NOT 98\n");
                 printf("Data.Vegcode has a code for water and the fuel model is NOT 98 -- IgnitionPoints()\n");
                                                  break;
        case 55: sprintf(ErrorMessage, "%s", "Something wrong with the number of Rows and Columns in
FIREHIST.asc");
                 printf("Something wrong with the number of Rows and Columns in FIREHIST.asc\n");
                          break;
        case 56: sprintf(ErrorMessage, "%s", "Something wrong with the X and Y origins in FIREHIST.asc\n");
                 printf("Something wrong with the X and Y origins in FIREHIST.asc\n");
                                  break;
        case 57: sprintf(ErrorMessage, "%s", "Something wrong trying to Run and Fill data with predicted
FLAMMAP\n");
                 printf("Something wrong trying to Run and Fill data with predicted FLAMMAP\n");
                          break;
        case 58: sprintf(ErrorMessage, "%s", "Something wrong trying to output the PredictedFlammapFT values\n");
                 printf("Something wrong trying to output the PredictedFlammapFT values\n");
                                  break;
        case 59: sprintf(ErrorMessage, "%s", "Appears to be a mismatch between Veg and Stage in 1st four
categories\n");
                 printf("Appears to be a mismatch between Veg and Stage in 1st four categories\n");
                          break;
        case 60: sprintf(ErrorMessage, "%s", "In InputPremoData there is a treelist of 209 and IS NOT water,
barren, etc.\n");
                 printf("In InputPremoData there is a treelist of 209 and IS NOT water, barren, etc.\n");
        break;
        case 61: sprintf(ErrorMessage, "%s", "Could not create and fill the AllBigTrees array in
Optimize.cpp\n");
                 printf("Could not create and fill the AllBigTrees array in Optimize.cpp\n");
                          break;
        case 62: sprintf(ErrorMessage, "%s", "The Landscape Goal value in globals.h is not
valid(FillValueToOptimize())!\n");
                 printf("The Landscape Goal value in globals.h is not valid!(FillValueToOptimize())\n");
                 break;
        case 63: sprintf(ErrorMessage, "%s", "In OutputForestDistribution() a cell has NONFOREST treelist but not
NONFOREST vegcode\n");
                 printf("In OutputForestDistribution() a cell has NONFOREST treelist but not NONFOREST
vegcode\n");        break;
        case 64: sprintf(ErrorMessage, "%s", "In OutputForestDistribution() problems figuring a VegCode from
PREMO values\n");
                 printf("In OutputForestDistribution() problems figuring a VegCode from PREMO values\n");
        break;
        case 65: sprintf(ErrorMessage, "%s", "In OutputForestDistribution() problems figuring a StageCode from
PREMO values\n");
                 printf("In OutputForestDistribution() problems figuring a StageCode from PREMO values\n");
                 break;
        case 66: sprintf(ErrorMessage, "%s", "Problem opening the current binary file\n");
                 printf("Problem opening the current binary file\n");
                                                  break;
        case 67: sprintf(ErrorMessage, "%s", "Problems generating in the MakeLink() function - undetermined\n");
                 printf("Problems generating in the MakeLink() function - undetermined\n");
                                          break;
        case 68: sprintf(ErrorMessage, "%s", "Not set up to handle more than X number of input landscape files -
see #define FILES\n");
                 printf("Not set up to handle more than X number of input landscape files - see #define
FILES\n");          break;
        case 69: sprintf(ErrorMessage, "%s", "ROWS and COLUMNS wrong in binary file - undetermined which
file\n");
                 printf("ROWS and COLUMNS wrong in binary file - undetermined which file\n");
                                  break;
        case 70: sprintf(ErrorMessage, "%s", "XLL and YLL wrong in binary file - undetermined which file\n");
                 printf("XLL and YLL wrong in binary file - undetermined which file\n");
                                  break;
        case 71: sprintf(ErrorMessage, "%s", "BYTEORDER wrong in binary file - undetermined which file\n");
                 printf("BYTEORDER wrong in binary file - undetermined which file\n");
                          break;
        case 72: sprintf(ErrorMessage, "%s", "There are more than 4 periods marked as TRUE in
EvaluateThisPeriod[]\n");
                 printf("There are more than 4 periods marked as TRUE in EvaluateThisPeriod[]\n");
                                  break;
        case 73: sprintf(ErrorMessage, "%s", "Problems in FillAvgInitialGoal6()\n");
                 printf("Problems in FillAvgInitialGoal6()\n");
                                                  break;
        case 74: sprintf(ErrorMessage, "%s", "In GetSumBigTrees() - encountered a duplicate Treelist value -
should not!\n");
                 printf("In GetSumBigTrees() - encountered a duplicate Treelist value - should not!\n");
                          break;
```

```
        case 75: sprintf(ErrorMessage, "%s", "Couldn't find a \"Key\" during bsearch() in FillPremoData() or
FillValueToOptimize() or InputNewPremoStandData()\n");
                 printf("Couldn't find a \"Key\" during bsearch() in FillPremoData() or FillValueToOptimize() or
InputNewPremoStandData()\n"); break;
        case 76: sprintf(ErrorMessage, "%s", "Something went wrong in the ApplyInsectDisturbance routine\n");
                 printf("Something went wrong in the ApplyInsectDisturbance routine\n");
                 break;
        case 77: sprintf(ErrorMessage, "%s", "WARNING....Count and Record do not match when determining Total,
insect, or fire occurances\n");
                 printf("WARNING....Count and Record do not match when determining Total, insect, or fire
occurances\n");                                  break;
        case 78: sprintf(ErrorMessage, "%s", "In FillValueToOptimize there is a mismatch of RecordNo and POT *
NP!!!!\n");
                 printf("In FillValueToOptimize there is a mismatch of RecordNo and POT * NP  !!!!\n");
                          break;
        case 79: sprintf(ErrorMessage, "%s", "In FillValueToOptimize there is a mismatch of ITL and count
!!!!\n");
                 printf("In FillValueToOptimize there is a mismatch of ITL and count !!!!\n");
                 break;
        case 80: sprintf(ErrorMessage, "%s", "Couldn't find a \"Key\" during bsearch() GetBaselineVTO(),
AdjustValuesSwap*(),
InitialSolutionGoal1(),GetSumBigTrees,Fill_SEra_PValues(),OutputBigTrees(),CalculateSumPeriodEra()\n");
                 printf("Couldn't find a \"Key\" during bsearch() GetBaselineVTO() or AdjustValuesSwap*(),
InitialSolutionGoal1(),GetSumBigTrees,Fill_SEra_PValues(), OutputBigTrees(),CalculateSumPeriodEra()\n");
        break;
        case 81: sprintf(ErrorMessage, "%s", "There are more subwatersheds than the global variable
MAX_SUBWATERSHEDS\n");
                 printf("There are more subwatersheds than the global variable MAX_SUBWATERSHEDS\n");
        break;
        case 82: sprintf(ErrorMessage, "%s", "Problems determining the eligible cells in
DetermineEligibleCells()\n");
                 printf("Problems determining the eligible cells in DetermineEligibleCells()\n");
        break;
        case 83: sprintf(ErrorMessage, "%s", "Problems filling the solution during FillSolution()-maybe a goal
not defined!\n");
                 printf("Problems filling the solution during FillSolution()-maybe a goal not defined!\n");
        break;
        case 84: sprintf(ErrorMessage, "%s", "Showing there are 0 sub-watersheds after
CountSolutionWatersheds()\n");
                 printf("Showing there are 0 sub-watersheds after CountSolutionWatersheds()\n"); break;
        case 85: sprintf(ErrorMessage, "%s", "Mismatch with the #of total or cells or #of Sub-watersheds in
Fill_SEra() or Fill_PerValues()\n");
                 printf("Mismatch with the #of total or cells or #of Sub-watersheds in Fill_SEra() or
Fill_PerValues()\n");          break;
        case 86: sprintf(ErrorMessage, "%s", "In CalculateSumPeriodEra() there was a problem\n");
                 printf("In CalculateSumPeriodEra() there was a problem\n"); break;
        case 87: sprintf(ErrorMessage, "%s", "X random initial solutions failed - try changing the SubEra[]
values\n");
                 printf("%d random initial solutions failed - try changing the SubEra[] values\n",INITIAL_TRYS);
        break;
        case 88: sprintf(ErrorMessage, "%s", "Couldn't find a \"Key\" during bsearch() in CompareEraMinor() \n");
                 printf("Couldn't find a \"Key\" during bsearch() in CompareEraMinor() \n");     break;
        case 89: sprintf(ErrorMessage, "%s", "No eligible cells found for this goal - something wrong\n");
                 printf("\a\a\a\a\a\a\aNo eligible cells- something wrong!!!\n");      break;
        case 90: sprintf(ErrorMessage, "%s", "The number of unique combinations of variables for
Apply[Insect][Fire][Total]Disturbance do not match!\n");
                 printf("\a\aThe number of unique combinations of variables for
Apply[Insect][Fire][Total]Disturbance do not match!\n");    break;
        case 91: sprintf(ErrorMessage, "%s", "Cellids and or Treelist values not matching in UpdateDataTreelist()
or UpdateDataWithNewStandData()\n");
                 printf("\a\aCellids and or Treelist values not matching in UpdateDataTreelist() or
UpdateDataWithNewStandData()\n");       break;
        case 92: sprintf(ErrorMessage, "%s", "Showing DougFir as TRUE with Pag series that should be
FALSE(ApplyInsectSeverity)\n");
                 printf("\a\aShowing DougFir as TRUE with Pag series that should be
FALSE(ApplyInsectSeverity)\n");         break;
        case 93: sprintf(ErrorMessage, "%s", "Received an invalid Pag value in one of the Insect Mortality
functions\n");
                 printf("\a\aReceived an invalid Pag value in one of the Insect Mortality functions\n");
        break;
        case 94: sprintf(ErrorMessage, "%s", "FillSubEraValues failed - probably have more than periods defined
than it knows how to fill\n");
                 printf("\a\aFillSubEraValues failed - probably have more than periods defined than it knows how
to fill\n");        break;
        case 95: sprintf(ErrorMessage, "%s", "Couldn't find a \"Key\" during bsearch() in
UpdateDataWithNewStandData()\n");
                 printf("Couldn't find a \"Key\" during bsearch() in UpdateDataWithNewStandData()\n");
        break;
        case 96: sprintf(ErrorMessage, "%s", "Records and RecordsNeeded not matching while creating new Premo
Stand Data Inv structures\n");
                 printf("Records and RecordsNeeded not matching while creating new Premo Stand Data Inv
structures\n");     break;
        case 97: sprintf(ErrorMessage, "%s", "Mismatch of numbers when counting and filling data for AllInfo[] or
UniqueList[]\n");
                 printf("Mismatch of numbers when counting and filling data for AllInfo[] or UniqueList[]\n");
        break;
        case 98: sprintf(ErrorMessage, "%s", "Couldn't find an initial treelist in the TreeIndex.txt file during
Initial FuelEra stuff\n");
                 printf("Couldn't find an initial treelist in the TreeIndex.txt file during Initial FuelEra
stuff\n");          break;
        case 99: sprintf(ErrorMessage, "%s", "Record count not matching when trying to fill up TREELIST_RECORDS
stuff\n");
                 printf("Record count not matching when trying to fill up TREELIST_RECORDS stuff\n");
        break;
        case 100: sprintf(ErrorMessage, "%s", "Got an unrecognizable VegClass during InitalizeFuelLoadings()\n");
```

```
                    printf("Got an unrecognizable VegClass during InitalizeFuelLoadings()\n");      break;
          case 101: sprintf(ErrorMessage, "%s", "Couldn't find a \"Key\" during bsearch() in
LoadInitialFuelModels()\n");
                    printf("Couldn't find a \"Key\" during bsearch() in LoadInitialFuelModels()\n");
          break;
          case 102: sprintf(ErrorMessage, "%s", "Found a mismatch of NONFOREST fuel models during a Period run of
FuelDecayAndContribution()\n");
                    printf("Found a mismatch of NONFOREST fuel models during a Period run of
FuelDecayAndContribution()\n");         break;
          case 103: sprintf(ErrorMessage, "%s", "In CalculateSumPeriodEra() a Cellid is showing up whose parent
Subwateshed was not in S_Era[]\n");
                    printf("In CalculateSumPeriodEra() a Cellid is showing up whose parent Subwateshed was not in
S_Era[]\n");        break;
          case 104: sprintf(ErrorMessage, "%s", "The CS.MaxGoal and CS.Goal values do not match for a SG_FIRE cell
- and they should\n");
                    printf("The CS.MaxGoal and CS.Goal values do not match for a SG_FIRE cell - and they
should\n");        break;
          case 105: sprintf(ErrorMessage, "%s", "The current stand goal assignment got reselected during a
neigborhood search - should not!\n");
                    printf("The current stand goal assignment got reselected during a neigborhood search - should
not!\n"); break;
          case 106: sprintf(ErrorMessage, "%s", "The current MoveObj value is not matching what was calculated
earlier in TestObj[], they should!\n");
                    printf("The current MoveObj value is not matching what was calculated earlier in TestObj[],
they should!\n");   break;

          default: sprintf(ErrorMessage, "%s", "Not sure what the heck the problem is!");
                         break;
          }

//write out the ErrorMessage
          fprintf(WriteOut,"%s",ErrorMessage);

//close the file
          fclose(WriteOut);


//Now exit the program
          exit(0);

}//end of Bailout

//*************************************************************************************
void EnvScope(int Eval[NP])
//*************************************************************************************
{
/*This function is to make a text file called ...\model\outputs\final_maps\envt.txt  that will
contain five lines - see below.
*/

FILE *OpenWrite;
int r, count, a;
char Temp[150];
int SubEra[NP];
char output[10];
//-------------------------------- end of variable defining ------------------------------

//****************************** APPLEGATE ***************************************************************************
#ifdef APPLEGATE_PROJECT
#if !defined(WHOLE_RUN) && !defined(COMPARE_RUN) && !defined(TINY_RUN) && !defined(LITTLE_RUN)
          printf("Environment scope not properly defined in Applegate_Globals.h...bailing\n");
          exit(0);
#endif
#ifdef WHOLE_RUN
          sprintf(output, "%s",ENVT);
          printf("\n\t\t\t********* USING THE ENTIRE APPLEGATE ENVIRONMENT **********\n\n\n");
#endif

#ifdef LITTLE_RUN
          sprintf(output, "%s",ENVT);
          printf("\n\t\t\t********* USING THE LITTLE APPLEGATE ENVIRONMENT **********\n\n\n");
#endif

#ifdef COMPARE_RUN
          sprintf(output, "%s",ENVT);
          printf("\n\t\t\t********* USING THE COMPARE ENVIRONMENT **********\n\n\n");
#endif

#ifdef TINY_RUN
          sprintf(output, "%s",ENVT);
          printf("\n\t\t\t********* USING THE TINY ENVIRONMENT **********\n\n\n");
#endif
#endif //#ifdef APPLEGATE



//****************************** FRAMEWORK ***************************************************************************
#ifdef FRAMEWORK_PROJECT
#if !defined(ELTA)
          printf("Environment scope not properly defined in Framework_Globals.h...bailing\n");
          exit(0);
#endif


#ifdef ELTA
```

```
          sprintf(output, "%s",ENVT);
          printf("\n\t\t\t********* USING THE ELDORADO-TAHOE N.F. ENVIRONMENT **********\n\n\n");
#endif
#endif //#ifdef FRAMEWORK_PROJECT


//=============================================================================================
//                                    Create and write out info to the file
//=============================================================================================

//Create and open the file
sprintf(Temp,"%s%s\\Envt.txt",PREFIX,MapDir);
OpenWrite = fopen(Temp, "w");                    //open in write mode

// ************* LINE 1 *****************
fprintf(OpenWrite,"%s\n",output);                                          //put in the "environment code" for
the amls to use

// ************* LINE 2 *****************
//NEW: 5 Nov:  Add in a second line that is used as a toggle for the AML in determining if it should
//create new ASCII files or new BINARY files for the initial landscape data
if(FILE_TYPE == 1)
                    fprintf(OpenWrite,"1\n");                          //create ASCII files
else
                    fprintf(OpenWrite,"2\n");                          //create BINARY files

// ************* LINE 3 *****************
//Third line will now have the GOAL_TO_USE in this file instead of the old way of putting in a separte goal.txt
file
fprintf(OpenWrite, "%d\n",GOAL_TO_USE);

// ************* LINE 4 *****************
//Now add in a line that has four values - each representing a period that we want to evaluate/map etc.. These
//values are from the Eval[] array passed in to this function
count=0;
for(r=0;r<NP;r++)
{
          if(Eval[r] == TRUE)
          {
                    count++;
                    if(count > 4)
                              Bailout(72);
                    else
                    fprintf(OpenWrite, "%d   ",r+1);
          }
}
fprintf(OpenWrite, "\n");                                                  //New line because
ArcInfo needs it

// ************* LINE 5 *****************
// The fifth line will have all the period ERA thresholds on the same line separated by at least a space
//Set and error checker because this isn't set up for more than 8 periods right now
if(NP > 8 )
          printf("Will also need to change stuff in the EnvScope() function to handle more than 8 periods\n");

//first initialize the SubEra array
for(a=0;a<NP;a++)
          SubEra[a] = 0;

//using the globals.h #define PER1,2,3,4_ERA put the threshold in the SubEra[] array
SubEra[0] = PER1_ERA;
SubEra[1] = PER2_ERA;
SubEra[2] = PER3_ERA;
SubEra[3] = PER4_ERA;
SubEra[4] = PER5_ERA;
SubEra[5] = PER6_ERA;
SubEra[6] = PER7_ERA;
SubEra[7] = PER8_ERA;

for(r=0;r<NP;r++)
   fprintf(OpenWrite, "%d  ", SubEra[r]);
fprintf(OpenWrite, "\n");                                                  //New line because
ArcInfo needs it


//Close the file
fclose(OpenWrite);

}//end EnvScope


//*****************************************************************************************
int WeatherStatus(int Weather[NP])
//*****************************************************************************************
{
//Fills up the Weather[] array with codes to use for Weather Type:
//        1 = Wet,  2 = Moderate,       3 = Mild Drought,    4 = Severe Drought
// Will always ouput these codes to ..\per0\weather.txt file so I can reenter the same
// weather pattern if I am Re-Running a simulation

//NOTE: added do loop to make sure there was at least one drought period during a simulation

int a;
char WeatherFile[150];
sprintf(WeatherFile, "%s%s%d\\per0\\weather.txt",PREFIX,INPUTS,GOAL_TO_USE);
```

```
//------------------------------------------- End of variable defining ----------------------------------------
------

//First, initialize the array - regardless if RERUN_SIM
for(a=0;a<NP;a++)
          Weather[a] = 0;


#ifndef RERUN_SIM

int rnd, Current, Continue, PreviousWeather, HadSevere = FALSE, AtLeastOneDrought;
FILE *WriteOut;

          //Now fill up the array.  If there are Two drought periods in a row, then it is a SEVERE drought on the
          //second occurance, and that can happen only ONCE during the entire simulation
          do
          {
                    //Reset this for each do loop try
                    PreviousWeather             = FALSE;
                    AtLeastOneDrought   = FALSE;

                    for(a=0;a<NP;a++)
                    {
                              do
                              {
                                        rnd = ( rand() % 100 + 1) ;

                                        //printf("RND in WeatherStatus is:  %d\n",rnd);

                                        if (rnd <= 25)                                                    // a 25% chance
that it IS a drought period
                                                  Current = 3;
                                        else if(rnd > 25 && rnd <= 90)
                                                  Current = 2;                                                        //65%
chance of a moderate period
                                        else
                                                  Current = 1;                                                        //10%
chance of a wet period

                                        //If this is a drought, then see if previous period was also a drought
                                        if( Current == 3)
                                        {
                                                  //Set the toggle to exit the big do loop - drought must be in 1st
4 periods
                                                  if( a < 4 )
                                                            AtLeastOneDrought = TRUE;

                                                  if( PreviousWeather == 3)               //Yes, 2 droughts in a
row == SEVERE drought, but can only occur once!
                                                  {
                                                            if(HadSevere == FALSE)                    //this is first
SEVERE - ok, accept
                                                            {
                                                                      Weather[a] = 4;                             //New
code to indicate a severe drought
                                                                      Continue = TRUE;
                                                                      PreviousWeather = Current;
                                                                      HadSevere = TRUE;
                                                            }
                                                            else
          //otherwise, pick another weather for this period
                                                                      Continue = FALSE;
                                                  }
                                                  else
          //previous period was not a drought, so this is ok, accept
                                                  {
                                                            Weather[a] = Current;
                                                            Continue = TRUE;
                                                            PreviousWeather = Current;
                                                  }
                                        }
                                        else
          //Is a Moderate or Wet year, accept and get next period
                                        {
                                                  Weather[a] = Current;
                                                  PreviousWeather = Current;
                                                  Continue = TRUE;
                                        }

                              }while(Continue == FALSE);

                    }//end for(a=0;a<NP;a++)
          }while(AtLeastOneDrought == FALSE);

          //Now write the current schedule out to a text file to use if the next simulation run is a RERUN_SIM
          WriteOut = fopen(WeatherFile, "w");                                       //no error checking

          for(a=0;a<NP;a++)
                    fprintf(WriteOut, "%d\n",Weather[a]);

          fclose(WriteOut);
#endif
```

```
#ifdef RERUN_SIM
          FILE *ReadIn;

          //open the weather file, read in the weather and store them in Weather[]
          ReadIn =  fopen(WeatherFile, "r");                                    //no error checking

          for(a=0;a<NP;a++)
                    fscanf(ReadIn, "%d", &Weather[a]);

          fclose(ReadIn);
#endif


return TRUE;

}//end WeatherStatus()

//*****************************************************************************
int EvaluatePeriods(int Eval[NP])
//*****************************************************************************
{
//This function will look at the four #define PER1,2,3,4 variables and set an appropriate
//flag of value '1' in the Eval[] array - which is used throughout program to determine
//whether or not to do things.  Mostly used for the periods will be mapped.

int a;

//first initialize the incoming array
for(a=0;a<NP;a++)
          Eval[a] = 0;


//using #define PER1,2,3,4 put a flag in the Eval[] array
Eval[PER1-1] = 1;                                                               //need correct array notation
Eval[PER2-1] = 1;
Eval[PER3-1] = 1;
Eval[PER4-1] = 1;

/*
for(a=0;a<NP;a++)
{
          if(Eval[a] == 1)
                    printf("Period %d (year %d) will be evaluated\n",a+1, (a+1)*5 );
}
*/

          return TRUE;
}//end EvaluatePeriods


//*****************************************************************************
int FillSubEraThresholds(int SubEra[])
//*****************************************************************************
{
/*
This function will fill the incoming SubEra array with the thresholds defined in globals.h
When checking the subwatershed ERA threshold constraint during the landscape optimization,
this will allow a higher threshold in earlier periods and the ability to change that
threshold as time goes on.
*/

int a;

//------------------------------ End of variable defining --------------------------------

//Set and error checker because this isn't set up for more than 8 periods right now
if(NP > 8 )
          return FALSE;

//first initialize the incoming array
for(a=0;a<NP;a++)
          SubEra[a] = 0;



//using the globals.h #define PER1,2,3,4_ERA put the threshold in the SubEra[] array
SubEra[0] = PER1_ERA;
SubEra[1] = PER2_ERA;
SubEra[2] = PER3_ERA;
SubEra[3] = PER4_ERA;
SubEra[4] = PER5_ERA;
SubEra[5] = PER6_ERA;
SubEra[6] = PER7_ERA;
SubEra[7] = PER8_ERA;


          return TRUE;
}//end FillSubEraValues

//*****************************************************************************
int CountSubWatersheds(int UM[])
//*****************************************************************************
{
/*
```

```
Whenever called up, it does 2 things:
          1) Fills the global UM[] array with SubWatershed #'s
          2) Returns the number of unique subwatersheds

This function works without using the structures that are seen later in the program because
it is done so early - just after reading the data into the Data.* arrays.  However,
the above global array and the return value (set to USW in main() ) are useable by all
other functions.
*/
int b,c,next,there;
ushort *ptr_minor;
int *ptr_um;                              //ptr_um is pointer to UM[]

//---------------------------------- End of variable defining ----------------------------

printf("*** Going to count up the total # of sub-watersheds there are...this number will include GIS slivers and
water bodies ***\n");

//Initialize the UniqueMinor[] array.
for(c=0;c<MAX_SUBWATERSHEDS;c++)
          UM[c] = 0;

//pick out the unique numbers in Data.Minor
next = 0;
for(c=0;c<UNIQUE;c++)
{
          there = 0;
          ptr_minor=&Data.Minor[c];

          if(*ptr_minor == 0)
                    break;                                        //assumes Data.Minor was initialized with 0's and there
are
                                                                            //no actual Minor sub-watershed values of 0

          for(b=0;b<MAX_SUBWATERSHEDS;b++)
          {
                    ptr_um = &UM[b];
                    if( (*ptr_minor) == (*ptr_um) )
                    {
                              there = 1;
                              break;                              //means the value is already in the UniqueMinor[] array
                    }
          }

          if(there == 0)
          {

                    UM[next] = *ptr_minor;
                    next++;
          }

}


if(next > MAX_SUBWATERSHEDS)
          Bailout(81);


printf("!!! There were %d UNIQUE sub-watersheds found...HOWEVER, they may not all be in the solution\n",next);

return next;

} //end of CountSubWatersheds


//*****************************************************************************************
void StartPeriodInfo(int p)
//*****************************************************************************************

{
          //A little screen notice that the period has starte and everything has checked out OK

          puts("\n\n\t\t*************************************************************");
          puts("\t\t***                                                       ***");
          printf("\t\t***           PERIOD %d is starting - Buckle UP!      \t   ***\n",p);
          puts("\t\t***                                                       ***");
          puts("\t\t*************************************************************");
}

//*****************************************************************************************
void EndPeriodInfo(int p)
//*****************************************************************************************

{
          //A little screen notice that the period has ended and everything has checked out
          //OK and program is continuing on to next period

          puts("\n\n\t\t*************************************************************");
          puts("\t\t***                                                       ***");
          printf("\t\t***           PERIOD %d has ended - continuing      \t   ***\n",p);
          puts("\t\t***           to next period if applicable.               ***");
          puts("\t\t***                                                       ***");
          puts("\t\t*************************************************************");
}
```

```
//*****************************************************************************
void DeleteOldStuff(void)
//*****************************************************************************
{
/*This will simply delete everything out of the \outputs\prescriptions\modeled\* directory
so only new prescription data (for this run of Safe) will be in there. */

char DeleteOldPrescriptions[250];
char DeleteOldStandData[250];
char JunkFile[300];

//Make the JunkFile which screen outputs can be redirected to
sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,ErrorDir);


//Make the strings for the system call
sprintf(DeleteOldPrescriptions, "del %s%s\\*.txt > %s",PREFIX,ModeledPresDir,JunkFile);
sprintf(DeleteOldStandData,             "del %s%s\\*.txt > %s",PREFIX,ModeledStandDataDir,JunkFile);


puts("=============================================================================================");
printf("Getting ready to delete all the files currently in:...\\outputs\\prescriptions\\modeled\\*\n");
printf("and those in...\\outputs\\StandData\\modeled\\*\n");
puts("=============================================================================================");

system(DeleteOldPrescriptions);
system(DeleteOldStandData);
}//end of DeleteOldStuff

//*****************************************************************************
void DeleteInitialStuff(void)
//*****************************************************************************
{
/*This will simply delete everything out of the \outputs\prescriptions\initial\* directory
so only new prescription data (for this run of Safe) will be in there. */

char DeleteOldPrescriptions[250];
char DeleteOldStandData[250];
char JunkFile[300];

//Make the JunkFile which screen outputs can be redirected to
sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,ErrorDir);


//Make the strings for the system call
sprintf(DeleteOldPrescriptions, "del %s%s\\*.txt > %s",PREFIX,InitialPresDir,JunkFile);
sprintf(DeleteOldStandData,             "del %s%s\\*.txt > %s",PREFIX,InitialStandDataDir,JunkFile);


puts("=============================================================================================");
printf("Getting ready to delete all the files currently in:...\\outputs\\prescriptions\\Initial\\*\n");
printf("and those in...\\outputs\\StandData\\Initial\\*\n");
puts("=============================================================================================");

system(DeleteOldPrescriptions);
system(DeleteOldStandData);
}//end of DeleteInitialStuff


//*****************************************************************************
void CleanAndSave(int Per, int Program, int Status)
//*****************************************************************************
{
/* This function will be called up after running FLAMMAP and FARSITE to clean up files and
save those that are necessary.

12Nov:  Not actually doing any saving yet!
*/

//Files that will potentially be used
char DelBlcFile[256];
char DelCbdFile[256];
char DelHeightFile[256];
char DelFuelFile[256];
char DelClosureFile[256];

//Create the appropriate filenames
sprintf(DelBlcFile,           "del %s%s%d\\per%d\\blc.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(DelCbdFile,           "del %s%s%d\\per%d\\cbd.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(DelHeightFile,        "del %s%s%d\\per%d\\height.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(DelFuelFile,          "del %s%s%d\\per%d\\fuel.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(DelClosureFile, "del %s%s%d\\per%d\\closure.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);


//During simulation period - delete the landscape ASCII files after Flammap uses - Farsite will use the
"layers.far" binary file!
if(Program == FLAMMAP && Status == ACTUAL)
{
          system(DelBlcFile);
          system(DelCbdFile);
          system(DelHeightFile);
          system(DelFuelFile);
          system(DelClosureFile);
}
```

```
//Call up the DeleteFar() function after FARSITE is finished for an ACTUAL run - the Layers.far file is a binary
//file created by Finney's program and has all the landscape data - it is huge and we don't need except during
//the period.
if(Program == FARSITE && Status == ACTUAL)
{
          DeleteFar(Per);

          if(EvaluateThisPeriod[Per-1] == FALSE)
          {
                    system(DelBlcFile);                                        //moving these here to see if it
stops Farsite from stalling
                    system(DelCbdFile);
                    system(DelHeightFile);
                    system(DelFuelFile);
                    system(DelClosureFile);
          }
}

}//end CleanAndSave

//*****************************************************************************
void DeleteModified(void)
//*****************************************************************************
{
/*This will simply delete everything out of the \outputs\prescriptions\Modified\* directory
so only new treelist data during a period will be in there.
*/

char DeleteMod[256];

sprintf(DeleteMod, "del %s%s\\*.txt",PREFIX,P_ModDir);


puts("\n=============================================================================================================
");
printf("Getting ready to delete all the files currently in ...\\model\\outputs\\prescriptions\\Modified\\*.txt\n");
puts("=========================================================================================================")
;

system(DeleteMod);


}//end of DeleteModified

//*****************************************************************************
void DeleteToModify(void)
//*****************************************************************************
{
/*This will simply delete everything out of the \outputs\prescriptions\ToModify\* directory
so only new treelist data during a period will be in there.
*/

char Delete[256];

sprintf(Delete, "del %s%s\\*.txt",PREFIX,P_ToModDir);


puts("\n=============================================================================================================
");
printf("Getting ready to delete all the files currently in ...\\model\\outputs\\prescriptions\\ToModify\\*.txt\n");
puts("=========================================================================================================")
;

system(Delete);


}//end of DeleteToModify


//*****************************************************************************
void MakeDirectories(void)
//*****************************************************************************
{
/*
This function is designed to check and make sure that all the directories that are going to
be used throughout the program exist already and if they don't exist then to create them -
otherwise, if the program tries to write a file out to a non-existent directory it bails.
*/
char TestDir[250];
int a,b;
//------------------------------ End of variable defining -------------------------------

//Always change to the appropriate DRIVE because I think some of these calls won't work across drive letters?
sprintf(TestDir, "%s",PREFIX);                                          //This directory has to be made by hand for
this code to work!!!!!
_chdir(TestDir);



//First make the main MODEL directory
sprintf(TestDir, "%s\\Model",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);
```

```
//*************************************************************************************************
//                                                                    Make all the directories under MODEL
//*************************************************************************************************
sprintf(TestDir, "%s\\Model\\amls",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\farsite",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\flammap",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\inputs",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\RerunData",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\SafeD",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\standopt",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);


//*************************************************************************************************
//                                                                    Make directories under Model\\amls
//*************************************************************************************************

sprintf(TestDir, "%s\\Model\\amls\\info",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

//*************************************************************************************************
//                                                                    Make directories under Model\\farsite
//*************************************************************************************************

sprintf(TestDir, "%s\\Model\\farsite\\farsite",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\farsite\\farsite\\Debug",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

//*************************************************************************************************
//                                                                    Make directories under Model\\flammap
//*************************************************************************************************

sprintf(TestDir, "%s\\Model\\flammap\\flammap",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\flammap\\flammap\\Debug",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

//*************************************************************************************************
//                                                                    Make directories under Model\\inputs
//*************************************************************************************************

sprintf(TestDir, "%s\\Model\\inputs\\Constant",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\inputs\\Constant\\info",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\inputs\\CommonInitial",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\inputs\\CommonInitial\\info",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

//-----For the goals under inputs
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
          sprintf(TestDir, "%s\\Model\\inputs\\goal%d",PREFIX,a);
          if( _chdir(TestDir)  )
                    _mkdir(TestDir);
```

```
        //For the periods under each goal
        for(b=0;b<=NP;b++)
        {
                sprintf(TestDir, "%s\\Model\\inputs\\goal%d\\per%d",PREFIX,a,b);
                if( _chdir(TestDir)  )
                        _mkdir(TestDir);

                sprintf(TestDir, "%s\\Model\\inputs\\goal%d\\per%d\\info",PREFIX,a,b);
                if( _chdir(TestDir)  )
                        _mkdir(TestDir);
        }

}


//*****************************************************************************************************
//                                                                  Make directories under Model\\outputs
//*****************************************************************************************************

//========================== DELUGE ==========================================
sprintf(TestDir, "%s\\Model\\outputs\\Deluge",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

//-----For the goals under Deluge
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
        sprintf(TestDir, "%s\\Model\\outputs\\Deluge\\goal%d",PREFIX,a);
        if( _chdir(TestDir)  )
                _mkdir(TestDir);
}

//========================== ERRORS ==========================================
sprintf(TestDir, "%s\\Model\\outputs\\Errors",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

//========================== FINAL_MAPS ======================================
sprintf(TestDir, "%s\\Model\\outputs\\final_maps",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\final_maps\\PlotFiles",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\final_maps\\info",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

//========================== GeneralData =====================================
sprintf(TestDir, "%s\\Model\\outputs\\GeneralData",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

//-----For the goals under GeneralData
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
        sprintf(TestDir, "%s\\Model\\outputs\\GeneralData\\goal%d",PREFIX,a);
        if( _chdir(TestDir)  )
                _mkdir(TestDir);
}

//========================== PERIOD(S) =======================================
//-----For the goals under outputs
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
        sprintf(TestDir, "%s\\Model\\outputs\\goal%d",PREFIX,a);
        if( _chdir(TestDir)  )
                _mkdir(TestDir);

        //For the periods under each goal
        for(b=0;b<=NP;b++)
        {
                sprintf(TestDir, "%s\\Model\\outputs\\goal%d\\per%d",PREFIX,a,b);
                if( _chdir(TestDir)  )
                        _mkdir(TestDir);

                sprintf(TestDir, "%s\\Model\\outputs\\goal%d\\per%d\\info",PREFIX,a,b);
                if( _chdir(TestDir)  )
                        _mkdir(TestDir);
        }

}

//========================== POSTSIMDATA =====================================
sprintf(TestDir, "%s\\Model\\outputs\\PostSimData",PREFIX);
if( _chdir(TestDir)  )
        _mkdir(TestDir);

//-----For the goals under PostSimData
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
        sprintf(TestDir, "%s\\Model\\outputs\\PostSimData\\goal%d",PREFIX,a);
        if( _chdir(TestDir)  )
```

```
                    _mkdir(TestDir);

          sprintf(TestDir, "%s\\Model\\outputs\\PostSimData\\goal%d\\info",PREFIX,a);
          if( _chdir(TestDir)  )
                    _mkdir(TestDir);
}

//========================== PRESIMDATA ====================================
sprintf(TestDir, "%s\\Model\\outputs\\PreSimData",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

//-----For the goals under PreSimData
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
          sprintf(TestDir, "%s\\Model\\outputs\\PreSimData\\goal%d",PREFIX,a);
          if( _chdir(TestDir)  )
                    _mkdir(TestDir);

          sprintf(TestDir, "%s\\Model\\outputs\\PreSimData\\goal%d\\info",PREFIX,a);
          if( _chdir(TestDir)  )
                    _mkdir(TestDir);
}

//========================== PRESCRIPTIONS ==================================
sprintf(TestDir, "%s\\Model\\outputs\\Prescriptions",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\Prescriptions\\Initial",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\Prescriptions\\Modeled",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\Prescriptions\\Modified",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\Prescriptions\\ToModify",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);


//========================== RASTER_OUT ====================================
sprintf(TestDir, "%s\\Model\\outputs\\raster_out",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\raster_out\\info",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);



//========================== STANDDATA =====================================
sprintf(TestDir, "%s\\Model\\outputs\\StandData",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\StandData\\Initial",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\StandData\\Initial\\binary",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\StandData\\Modeled",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);



//========================== VECTOR_OUT ====================================
sprintf(TestDir, "%s\\Model\\outputs\\vector_out",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\outputs\\vector_out\\info",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);


//*************************************************************************************************
//                                                          Make directories under Model\\RerunData
//*************************************************************************************************
//-----For the goals under RerunData
for(a=1;a<=LANDSCAPE_GOALS;a++)
{
          sprintf(TestDir, "%s\\Model\\RerunData\\goal%d",PREFIX,a);
          if( _chdir(TestDir)  )
                    _mkdir(TestDir);
```

```
}


//*************************************************************************************************
//                                                                  Make directories under Model\\SafeD
//*************************************************************************************************

sprintf(TestDir, "%s\\Model\\SafeD\\SafeD",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\SafeD\\SafeD\\Debug",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);


//*************************************************************************************************
//                                                                  Make directories under Model\\standopt
//*************************************************************************************************

sprintf(TestDir, "%s\\Model\\standopt\\Premo",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);

sprintf(TestDir, "%s\\Model\\standopt\\Premo\\Debug",PREFIX);
if( _chdir(TestDir)  )
          _mkdir(TestDir);



}//end MakeDirectories

//*************************************************************************************************
void CopyExecutables(void)
//*************************************************************************************************
{
          //Copy some executables from their debug directory in the Applegate directories over to the Framework
char KillFile[256];
char CopyFile[256];
char JunkFile[300];

//Make the JunkFile which screen outputs can be redirected to
sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,ErrorDir);


//The PREMO executable
sprintf(KillFile, "del %s%s > %s",PREFIX,PremoProgName,JunkFile);
sprintf(CopyFile, "copy g:%s %s%s > %s",PremoProgName,PREFIX,PremoProgName,JunkFile);
system(CopyFile);

//The FLAMMAP executable
sprintf(KillFile, "del %s%s > %s",PREFIX,FlammapName,JunkFile);
sprintf(CopyFile, "copy g:%s %s%s > %s",FlammapName,PREFIX,FlammapName,JunkFile);
system(CopyFile);

//The FARSITE executable
//sprintf(KillFile, "del %s%s > %s",PREFIX,FarsiteName,JunkFile);
//sprintf(CopyFile, "copy g:%s %s%s > %s",FarsiteName,PREFIX,FarsiteName,JunkFile);
//system(CopyFile);

}//end CopyExecutables

DATA.H

/*
18 Aug 99 - I made this header file to store the definition for several variables
that were being used by all the *.cpp files.  Rather than put these at top of each
file I can update these right here, only once and they will all have the new definition
*/

//in Misc.cpp
extern void Bailout(int ErrorNumber);
void PrintToStat(int Line, ulong Value);

//in mgsort.cpp
extern int mgsort(void *data, int size, int esize, int i, int k, int (*compare)
   (const void *key1, const void *key2));


//An array to use for toggling whether to run certain portions at different period (1=yes, 0=no)
extern int EvaluateThisPeriod[NP];               //YES for periods ? (see misc.cpp)

//defined in ReadData.cpp
extern struct Main{
          ulong Cellid[UNIQUE];                                          //        Values for entire
Applegate watershed
          ushort GridRow[UNIQUE];
          ushort GridColumn[UNIQUE];
          ulong Treelist[UNIQUE];
          ushort Elev[UNIQUE];                                           //        in meters
          ushort Aspect[UNIQUE];
          ushort Slope[UNIQUE];
          ushort Goal[UNIQUE];
          ushort Owner[UNIQUE];
          ushort Pag[UNIQUE];
```

```
        ushort Alloc[UNIQUE];                                   //      Does NOT include an
allocation for stream buffers - use Data.Buffer
        ushort Minor[UNIQUE];
        ushort Hold[UNIQUE];
        ushort Buffer[UNIQUE];                                  //  Stream buffers on FED land
only!!  NODATAFLAG = noBuff, 100 = in Buffer
        ushort FireHistory[UNIQUE];                     //      Old fire perimeters
NODATAFLAG = not in, 100 = in old polygon
        ushort InitialVeg[UNIQUE];
        ushort InitialStage[UNIQUE];
        ushort PRule[UNIQUE];


        //Get calculated within SafeD
        ushort InitialDuff[UNIQUE];                             // These initial Fuel Loadings will be
divided by TONS - convert back by
        ushort InitialLitter[UNIQUE];                   // multiplying by TONS when using.(also using
FUEL_LOAD_EXP to make ushort)
        ushort InitialClass25[UNIQUE];
        ushort InitialClass1[UNIQUE];
        ushort InitialClass3[UNIQUE];
        ushort InitialClass6[UNIQUE];
        ushort InitialClass12[UNIQUE];
        ushort InitialClassOver12[UNIQUE];
        ushort InitialFuelModel[UNIQUE];
        ushort InitialEra[UNIQUE];
        ushort FuelModel[UNIQUE][NP];
        ushort Duff[UNIQUE][NP];                                //Divided by TONS - and multiply by
FUEL_LOAD_EXP
        ushort Flame[UNIQUE];                                   //Will hold the current flame
length interval from a FARSITE run

        //Data that will come from Premo
        ushort Basal[UNIQUE][NP];                               // converting...divide by 10 to get REAL
value
        ushort Closure[UNIQUE][NP];                             // Truncating to closest integer
        ushort CBDensity[UNIQUE][NP];                   // converting...divide by 100 to get REAL value
        ushort HLC[UNIQUE][NP];                                 // Truncating to closest integer
        ushort StandHeight[UNIQUE][NP];                         // Truncating to closest integer
        ushort BigTrees[UNIQUE][NP];                    // converting...divide by 10 to get REAL value
        ushort Era[UNIQUE][NP];                                 // converting...divide by 100 to
get REAL value
        ushort Vegcode[UNIQUE][NP];                             //
        float CFHarvest[UNIQUE][NP];                    //
        ushort  Litter[UNIQUE][NP];                             //-------  All these fuel loadings will be
divided (/) by TONS -- Premo * by TONS
        ushort  Class25[UNIQUE][NP];                            //-------  when it outputted and some are too
big to fit in ushort.  Convert
        ushort  Class1[UNIQUE][NP];                             //-------  when needed back to TONS.
        ushort  Class3[UNIQUE][NP];
        ushort  Class6[UNIQUE][NP];
        ushort  Class12[UNIQUE][NP];
        ushort  ClassOver12[UNIQUE][NP];
        } Data;

extern int link[ROWS][3];


struct PREMO_RECORD
{
        //Key data
        ulong   Treelist;
        ushort  Goal;
        ushort  Hold;
        ushort  Period;

        //Regular attribute data
        ushort  Basal;
        ushort  Closure;
        ushort  Density;
        ushort  HeightCrown;
        ushort  StandHeight;
        float   Rev;
        ushort  BigTrees;
        ushort  Vegcode;
        float   Harvest;

        //Fuel loading stuff
        ushort  Litter;
        ushort  Class25;
        ushort  Class1;
        ushort  Class3;
        ushort  Class6;
        ushort  Class12;
        ushort  ClassOver12;
};

struct PTR_PREMO_RECORD
{
        struct  PREMO_RECORD *CurrentSD;
        ulong   Records;
};
```

```
struct TREELIST_RECORD
{
          ushort    Plot;
          ushort    Status;
          float     Tpa;
          ushort    Model;
          ushort    Report;
          float     Dbh;
          float     Height;
          float     Ratio;
          ushort    Condition;

          //These may get calculated, but there should be one for each line in a treelist record
          float     Basal;
          float     CanopyWidth;
          float     Hlc;
};


struct OPTIMIZE_SINGLE_VALUE            //Holds data on a "Prescription" basis - neglects individual cell variation
{
          //Key data
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;

          //attribute data
          ushort    Value[NP];                              //This will have "rounded" values if using BigTrees or
other Float/Double data
          float  BigTrees[NP];
          ushort    Rev[NP];
          ushort    CFHarvest[NP];
};

struct SOLUTION                                               //Holds data on an individual cell basis
{
          //Key data
          ushort    Minor;
          ulong     Cellid;
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;
          ushort    InitialEra;
          ushort    MaxGoal;                        //For the framework - this is the Highest stand goal number
allowed to be chosen for a cell
          ushort    PeriodEra[NP];                  //These get updated as heuristic is running and at end should
have correct values for given solution
};

struct P_INFO
{
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;
};

struct ERA                                                          //Holds data on a SUBWATERSHED basis
{
          //Key data
          ushort Minor;

          //attribute data
          ulong     Count;
          ulong     SumInitialEra;                  //not really used - gets filled in FillS_Era
          ulong     SumPeriodEra[NP];
};


struct TREELIST_FOR_PREMO
{
          ulong     OldTreelist;
          ushort    Goal;
          ushort    Hold;
          ulong     NewTreelist;

};


struct HIT_BY_DISTURB
{
          //Key data
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;

          ushort    Pag;                //These 4 are only used for INSECTS
          ushort    DougFir;
          ushort    TrueFir;
          ushort    Pine;

          ushort  Interval;   //This is > 0 only when used for FIRE

          //regular attribute data
          ulong     Cellid;
          ulong     NewTreelist;
```

```
};

struct UNIQUE_INSECT
{
          //Key data
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;
          ushort    Pag;
          ushort    DougFir;
          ushort    TrueFir;
          ushort    Pine;

          //regular attribute data
          ulong     NewTreelist;
};


struct UNIQUE_FIRE
{         //Key data
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;
          ushort  Interval;

          //regular attribute data
          ulong     NewTreelist;
};

struct FOFEM_MATRIX
{
          double BO[21][8];
          double DF[21][8];
          double HW[21][8];
          double PP[21][8];
          double SP[21][8];
          double WF[21][8];
};

struct NEW_STAND_DATA                    //Part or All of these may get used at any time, depending on what's needed
{
          ulong     Treelist;
          float     Basal;
          ushort    VegClass;           //These 3 are part of our Veg-Structural classification              --
values 1-9
          ushort    Qmd;                          // values 0-6
          ushort    CoverClass;                   // 0 is <60%, 1 is >= 60%
          ushort    Closure;            //The % canopy closure
          ushort    Density;
          ushort    HeightCrown;
          ushort    StandHeight;

          float     BigTreesKilled;               //These are used post-disturbance to track how much damage is
actually happening
          float     BasalAreaKilled;
};

struct STAND_CLASS            //This is used a temporary holder of data to pass in to certain functions from
anywhere
{
          float     Basal;
          ushort    VegClass;           //These 3 are part of our Veg-Structural classification              --
values 1-9
          ushort    Qmd;                          // values 0-6
          ushort    CoverClass;                   // 0 is <60%, 1 is >= 60%
          ushort    Closure;            //The % canopy closure
          float     HeightCrown;

};

struct INITIAL_FUELS
{
          ulong     Treelist;
          ushort    Goal;
          ushort    Hold;

          double    Duff;                               //These are the different "pools" of wood to use
          double    Litter;
          double    Class25;                  // 0 - .25"
          double    Class1;                             // .25 - 1"
          double    Class3;                             // 1 - 3"
          double    Class6All;                          // 3 - 6" total of Part1 and Part2
          double    Class6Part1;              // 3 - 6" from:  crown lift, harvest crowns, background breakage
          double    Class6Part2;              // 3 - 6" from:  stump to dwd
          double    Class12;                  // 6 - 12"
          double    ClassOver12;              // Over 12"
          double    Hour1Fuels;                         //These are groupings that Jim and Bernie use to
actually define a fuel model
          double    Hour10Fuels;
          double    Hour100Fuels;


          double    MC_Duff;                            //"parallel" variables for when stand is MC > 3000'
          double    MC_Litter;
```

```
        double    MC_Class25;
        double    MC_Class1;
        double    MC_Class3;
        double    MC_Class6All;
        double    MC_Class6Part1;
        double    MC_Class6Part2;
        double    MC_Class12;
        double    MC_ClassOver12;
        double    MC_Hour1Fuels;
        double    MC_Hour10Fuels;
        double    MC_Hour100Fuels;

        float     Basal;
        ushort    VegClass;          //These 3 are part of our Veg-Structural classification                --
values 1-9
        ushort    Qmd;                        // values 0-6
        ushort    CoverClass;                 // 0 is <60%, 1 is >= 60%
        ushort    Closure;           //The % canopy closure
        ushort    FuelModel;
        ushort    MC_FuelModel;
};

struct CURRENT_ERAS
{
        struct    OPTIMIZE_SINGLE_VALUE *ptr_osv;
        float     NetEra[NP];
        float   CurrentEra;
        int               Cell;                     //to hold the cell array position value
        int               NeedsDecay;               //defaults to FALSE, check for TRUE when needed
};
```

PREMOSTUFF.CPP

```
//*************************************************************************************************************
// This PremoStuff.cpp file will hold the functions, etc. that are used in conjunction with
//   Heidi's Stand Optimizer program (PREMO).
//*************************************************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"


//These functions will control stuff for Time 0 - for the initial Premo runs
void CreateTreeIndex(void);
void InitialPremo(void);
void FillInitialPremoData(int per);
void CopyStandOpt(int Treelist, int Goal, int Hold);
void CreateSortedPremoBinaryFile(void);
int LookAtPremoRecords(const void *ptr1, const void *ptr2);

//These functions will control stuff for post-disturbance Premo runs (i.e. starting Time period 1)
int ManageNewPremoRuns(ulong FTL, int Per);
int CountTotalHit(ulong FTL);
int FillHitListForPremo( struct HIT_BY_DISTURB HitList[] , ulong FTL );
int FillForPremo( struct TREELIST_FOR_PREMO FP[], struct HIT_BY_DISTURB HitList[], int Count);
void MakeNewPremoRuns( struct TREELIST_FOR_PREMO FP[], int Count, int ActualPer);
void CreatePostDisturbanceStandDataStructure( struct TREELIST_FOR_PREMO FP[], int Count, int ActualPer,struct
PTR_PREMO_RECORD *ptr_info);
void InputNewPremoStandData( struct PTR_PREMO_RECORD *ptr_info, int ActualPer, ulong FTL, int HitCount);
int CompareHitListForNewPremo(const void *ptr1, const void *ptr2);
int CountUniqueNewPremoHits(struct HIT_BY_DISTURB HitList[], int Count);

//In Misc.cpp
extern void DeleteInitialStuff(void);

//in StandData.cpp
extern void NewStandHLC( struct STAND_CLASS *Stand  );

//A couple of globals to use for figuring out number of InitialTreeList and Potential prescriptions
ulong ITL;
//------------------------------- End of function definitions ----------------------------------


//*************************************************************************************************
int ManageNewPremoRuns(ulong FTL, int Per)
//*************************************************************************************************
{
/*
This function will control how end-of-period Premo runs are handled.  The same strategy as that used
for the individual disturbances will be employed.  Also, the same strategy used to fill up the
Data.*[] arrays with initial Premo data will also be employed.  Basically:

1) Figure out all the cells hit by any disturbance
2) Figure out UNIQUE combinations of variables that will affect how many Premo calls to make (this will
```

```
   vary, see NOTE: in code after FillHitListForPremo() is called up.
3) Store info on those unique combinations and call up Premo appropriately
4) Create storage for all Stand Data created by Premo
5) Fill the storage up appropriately
6) Finally, transfer that data back into the Data.*[] arrays


NOTE: keep in mind that there will be MANY treelist in the \modified\ directory that do NOT get
sent to Premo.  They may have been modified once by an early disturbance and then (assume that
only one cell had that particular treelist) let's say another disturbance in the same period hits
the same cell.  The updates will be done to the already modified treelist and a NEW treelist will
be generated - thus, the old one is no longer a factor in this period (unless another stand was also
using it and was NOT hit by another disturbance). In any case, this is all tracked but the result may
be an apparant loss of treelist being used for new Premo calls, but that is not the case.

*/

int a=0, HitCount, Records, Unique, Unique2;

struct PTR_PREMO_RECORD *ptr_info;      //will point to the "SdInfo" structure - for the memset & function calls
struct PTR_PREMO_RECORD SdInfo;
//---------------------------------- End of variable defining --------------------------------

//First count how many different stands were hit this period
HitCount = CountTotalHit(FTL);

//If there are no cells getting hit by any disturbances, then just return back to main
if( HitCount == FALSE )
{
          printf("!!! There were NO cells effected by disturbances this period - that is odd !!!\n");
          return TRUE;
}

//create an array of structures on the free store to hold info on all the different cells
struct HIT_BY_DISTURB (*HitList) = new struct HIT_BY_DISTURB[HitCount];
if( HitList == NULL )
          printf("Problems allocating memory for HitList[] with %d records\n",HitCount);

//Initialize
memset( HitList, 0, sizeof(struct HIT_BY_DISTURB) * HitCount);

//Fill up the array of HitList structures
Records = FillHitListForPremo(HitList, FTL);
if(Records != HitCount)
          Bailout(77);


/*
NOTE:  At this time, we are considering not re-optimizing the landscape prescription selection after a period is
over.  The
current strategy is to just re-use the current GOAL-HOLD assignment for each cell.  That has a HUGE effect on the
next
part - Sorting and Identifying unique combinations of cells that need new PREMO runs.  By not re-optimizing at the
landscape
level we can sort and identify unique combinations in HitList[] by Treelist - Goal - Hold.
In the future, if stand re-optimization takes place, new Compare*,CountUnique*, and Fill* type functions will have
to
be developed to correctly account for new strategy (e.g. sort and get unique Treelist-Interval combination only ).
*/
//sort those records by:  Treelist-Goal-Hold
printf("\nGetting ready to sort the stands by Treelist-Goal-Hold.....this will take awhile for %lu
cells\n\n",HitCount);


//Sort HitList by Treelist-Goal-Hold-Flame
mgsort( (void*)HitList,
          //base
                    HitCount,
          //count of records
                    sizeof( struct HIT_BY_DISTURB),                                                    //size of each
record
                    0,  HitCount-1,
          //current division ( always: 0, Count-1 )
                    CompareHitListForNewPremo   );
          //compare function


//Count up how many of those records in HitList are actually unique combinations of Treelist-Goal-Hold
//REMEMBER: May have to develop new function if landscape reoptimization occurs at end of each period
Unique = CountUniqueNewPremoHits(HitList,HitCount);
printf("!!!There were actually %d unique records that will each require a PREMO run\n",Unique);


//Create an array of structures to hold data pertaining to each unique T-G-H combination
struct TREELIST_FOR_PREMO (*ForPremo) = new struct TREELIST_FOR_PREMO[Unique];
if( ForPremo == NULL )
          printf("Problems allocating memory for ForPremo[] with %d records\n",Unique);
//Initialize
memset( ForPremo, 0, sizeof(struct TREELIST_FOR_PREMO) * Unique);


//Fill up the array of ForPremo structures and make sure same # of records processed
Unique2 = FillForPremo(ForPremo, HitList, HitCount);
if(Unique2 != Unique)
```

```
        Bailout(90);

//Send ForPremo off to have the Premo runs made
#ifdef END_PERIOD_PREMO
MakeNewPremoRuns(ForPremo, Unique, Per);
#endif


//Initialize a structure that will hold data pertaining to the new Stand Data "Inv" structure that will get create
ptr_info = &SdInfo;
memset(ptr_info, 0, sizeof(struct PTR_PREMO_RECORD) );


//Create a sorted structure to hold all the new StandData
CreatePostDisturbanceStandDataStructure(ForPremo,Unique,Per,ptr_info);


//Input all the new Stand Data into the Data.*[] arrays
InputNewPremoStandData(ptr_info,Per, FTL, HitCount);


//Now that we are done with the Stand Data Inv[] memory block, delete it!
delete [] ptr_info->CurrentSD;

//and delete other stuff on free store
delete [] HitList;
delete [] ForPremo;

return TRUE;

}//end ManageNewPremoRuns

//*****************************************************************************************************************
*************
void InputNewPremoStandData( struct PTR_PREMO_RECORD *ptr_info, int ActualPer, ulong FTL, int HitCount)
//*****************************************************************************************************************
*************
{
/*
This function will essentially work the same way as FillInitialPremoData() does except in deciding which CELLS to
fill.

1) Use same strategy as in CountTotalHit() to determine if a Data.*[] cell got hit.  IF SO then
2) Grab the current Treelist, Goal, and Hold, of that cell and make a "key"....
3) Use that key to search the sorted array of "Inv" structures and find the match
4) Once a match is found, fill up the appropriate Data.*[] arrays with data from the matching key!

The Inv structure should have been sorted in CreatePostDisturbanceStandDataStructure() before getting here
*/
int a,y;
int Count, ArrayPer;

//structures
struct PREMO_RECORD Key;
struct PREMO_RECORD *ptr_record;
struct PREMO_RECORD *Inv;

//------------------------------- End of variable defining ----------------------------------------------

puts("\n    ***********************************************************************************************");
printf("    ******  Updating the Data.*[][] arrays with new data from the prescription optimizer *******\n");
puts("    ***********************************************************************************************\n");

//Reassign ptr_info->CurrentSD to the new pointer Inv so it's easier to write and access
Inv = ptr_info->CurrentSD;


//Start going through the Data.*[] arrays and find those that were hit by a disturbance this period
Count = 0;
for(a=0;a<UNIQUE;a++)
{
        if( Data.Cellid[a] == FALSE )                                                    //done looking
through arrays
                break;

        if( Data.Treelist[a] >= FTL )                                                    //This cell WAS
hit by something
        {
                //Start to make some of the "Key" for this cell to use in looking for the correct record in the
array of Inv structures
                Key.Treelist                    = Data.Treelist[a];
                Key.Goal                        = Data.Goal[a];
                Key.Hold                        = Data.Hold[a];

                //make another loop to account for the period
                //*** IMPORTANT:  note this starts filling in data for the FOLLOWING period - not this period!
                for(y=ActualPer+1;y<=NP;y++)
                {
                        //Finish off the key with the actual search period
                        Key.Period = (ushort)y;

                        //Make the ArrayPer variable
                        ArrayPer = y-1;

                        //Now use bsearch to find the matching record in the array of PremoInv structures
```

```
                                ptr_record = (struct PREMO_RECORD*)bsearch(
                                        &Key,
                                        (void *)Inv,
                                        (size_t)ptr_info->Records,
                                        sizeof( struct PREMO_RECORD),
                                        LookAtPremoRecords  );

                                if(ptr_record == NULL)
                                        Bailout(75);

                                //Check to see if the Vegcode value is Mixed Conifer - if so it must be broken into >
or < 3000'
                                if( (int)(ptr_record->Vegcode / 100 ) == VC_MC )
                                {
                                        if( Data.Elev[a] >= (3000*FT2M) )
                                                                //It's over 3,000 ft
                                                Data.Vegcode[a][ArrayPer]               =       ptr_record-
>Vegcode + 500;                         //This will give it 10**
                                        else
                                                Data.Vegcode[a][ArrayPer]               =       ptr_record-
>Vegcode;                               //Leave as is
                                }
                                else
                                        Data.Vegcode[a][ArrayPer]                       =       ptr_record-
>Vegcode;                               //Leave as is


                                //Fill in the rest of Data.*[] arrays with the data accessible from the pointer
returned above
                                //Everyone should already have the proper type
                                Data.Basal[a][ArrayPer]                                 =       ptr_record->Basal;
                                Data.Closure[a][ArrayPer]               =       ptr_record->Closure;
                                Data.CBDensity[a][ArrayPer]             =       ptr_record->Density;
                                Data.HLC[a][ArrayPer]                                   =       ptr_record->HeightCrown;
                                Data.StandHeight[a][ArrayPer] =         ptr_record->StandHeight;
                                Data.BigTrees[a][ArrayPer]              =       ptr_record->BigTrees;
                                Data.CFHarvest[a][ArrayPer]             =       ptr_record->Harvest;

                                Data.Litter[a][ArrayPer]                =       ptr_record->Litter;
                                Data.Class25[a][ArrayPer]               =       ptr_record->Class25;
                                Data.Class1[a][ArrayPer]                =       ptr_record->Class1;
                                Data.Class3[a][ArrayPer]                =       ptr_record->Class3;
                                Data.Class6[a][ArrayPer]                =       ptr_record->Class6;
                                Data.Class12[a][ArrayPer]               =       ptr_record->Class12;
                                Data.ClassOver12[a][ArrayPer] =         ptr_record->ClassOver12;


                        }//end for(y=0 ... )

                        Count++;

                }//end if(Data.Treelist ... )
        }//end for(a=0 ... )

        //Error check that the correct number of cells updated - same checker is in ManageNewPremoRuns()
        if(Count != HitCount)
                Bailout(77);



}//end InputNewPremoStandData

//*****************************************************************************************************************
**********************
void CreatePostDisturbanceStandDataStructure(struct TREELIST_FOR_PREMO FP[], int Count, int ActualPer, struct
PTR_PREMO_RECORD *ptr_info)
//*****************************************************************************************************************
**********************
{
/*
This function will pretty much do the same thing as was done in the CreateSortedPremoBinaryFile()
function that was used at Time 0.  Basically, read and store all the data for all the new
SD_*_*_*.txt files that were just created after the Premo runs at the end of a period.  Then sort
that data appropriately so that another function can go through the Data.*[] arrays and find
those cells that were hit by disturbance(s) this period and then find their new stand data.

17 FEB 00 - adding modification to stand HLC and CBD calculations here because it is easier to have
the Inv[] structure below just have the correct values now rather than waiting.

REMEMBER:  at period NP, this will create "0" RecordsNeeded and thus will be skipped - Oh, back in main() this
function does NOT get called on last period anyways, duh.
*/
FILE *DataIn;
char Temp[256];

ulong RecordsNeeded, Record, Treelist;
ushort Goal, Hold;
int a,y,r=0;
int DataPeriod;
double RealBasal, RealClosure, RealCBD, RealHLC, RealHeight, RealRev, RealBigTrees, Harvest;
ushort VegCode;
double RealLitter, RealClass25, RealClass1, RealClass3, RealClass6, RealClass12, RealClassOver12;
double LoadFactor;
double ModCbd;
struct STAND_CLASS StandClass;
```

```
struct STAND_CLASS *ptr_stand;
ushort TempCode;
int TempVeg, TempDiam, TempCover;
//--------------------------- End of variable defining -------------------------------------

//Determine the actual # of records
//NOTE: May need to change below if re-optimizing at stand level to include records for each GOAL and HOLD comb.
also
RecordsNeeded = Count * (NP-ActualPer);

if(RecordsNeeded == FALSE)
  return;


//Create an array of structures on the free store to hold these
struct PREMO_RECORD (*Inv) = new struct PREMO_RECORD[RecordsNeeded];
if( Inv == NULL )
         printf("Problems allocating memory for Inv[] with %lu elements\n",RecordsNeeded*sizeof(PREMO_RECORD));

//Initialize
memset( Inv, 0, sizeof(struct PREMO_RECORD) * RecordsNeeded );


//*****************  Start pumping data into the array of Inv[] structures **********************

//Set the LoadFactor that will be used to convert the incoming FuelLoadings (which are in LBS) to TONS but
//keeping some precision by multiplying by FUEL_LOAD_EXP
LoadFactor          = TONS * FUEL_LOAD_EXP;

Record=0;
for(a=0;a<Count;a++)                                                                    //for each record in the
FP structure
{
         //Get the treelist, goal, and hold to be used
         Treelist  =         FP[a].NewTreelist;
         Goal                =         FP[a].Goal;
         Hold                =         FP[a].Hold;


         //Start by opening the correct SD_*_*_*.txt file ONCE - always in the "\modeled\" directory at end of
period
         sprintf(Temp, "%s%s\\SD_%d_%d_%d.txt",PREFIX,ModeledStandDataDir,Treelist,Goal,Hold);
         DataIn = fopen(Temp, "r");
         if (DataIn == NULL)
                  fprintf(stderr, "opening of %s failed: %s\n",Temp, strerror(errno));


         //The SD_*_*_*.txt files will have a line of data for the incoming treelist at and it is NOT needed.
         //SafeD works by assuming the the 2nd period listed is after Harvest activities and 5-yr growth
         fscanf(DataIn,"%d %lf %lf %lf %lf %lf %lf %lf %hu %lf %lf %lf %lf %lf %lf %lf %lf",   &DataPeriod,
&RealBasal,
                  &RealClosure, &RealCBD, &RealHLC, &RealHeight, &RealRev, &RealBigTrees, &VegCode, &Harvest,
                  &RealLitter, &RealClass25, &RealClass1, &RealClass3, &RealClass6, &RealClass12,
&RealClassOver12);


         //Now scan in all the data ONCE and store in the array of Inv structures
         //I assume that the SD_*_*_* have Period "ActualPer" + 1 on line 2
         for(y=ActualPer+1;y<=NP;y++)
         {
                  //First, scan in the lines from SD* - this current period data is "over" and we don't need this
                  fscanf(DataIn,"%d %lf %lf %lf %lf %lf %lf %lf %hu %lf %lf %lf %lf %lf %lf %lf %lf",
&DataPeriod, &RealBasal,
                           &RealClosure, &RealCBD, &RealHLC, &RealHeight, &RealRev, &RealBigTrees, &VegCode,
&Harvest,
                           &RealLitter, &RealClass25, &RealClass1, &RealClass3, &RealClass6, &RealClass12,
&RealClassOver12);

                  if(DataPeriod != y)
                  {
                           printf("problem with period values in the PREMO files for %lu %hu
%hu\n",Treelist,Goal,Hold);
                           getchar();                                  //make program pause here
                  }

                  //Check and make sure values that eventually will be converted to USHORT are OK in terms of
65535 thingy!!
                  if(RealBasal >= 6553  || RealCBD >= 655 || RealBigTrees >= 6553 || RealLitter >= 6553*TONS ||
                           RealClass25 >= 6553*TONS || RealClass1 >= 6553*TONS || RealClass3 >= 6553*TONS ||
                           RealClass6 >= 6553*TONS || RealClass12 >= 6553*TONS || RealClassOver12 >= 6553*TONS)
                  {
                           printf("Guilty is:  Tree %d  Goal %d and Hold %d\n", Treelist, Goal, Hold);
                           Bailout(49);
                  }

                  Inv[Record].Treelist                    =         (ulong)Treelist;
                  Inv[Record].Goal                        =         (ushort)Goal;
                  Inv[Record].Hold                        =         (ushort)Hold;
                  Inv[Record].Period                      =         (ushort)y;
                                             //This is the next period!
                  Inv[Record].Basal                       =         (ushort)(floor(RealBasal * BASAL_EXP));
                  Inv[Record].Closure                     =         (ushort)(floor(RealClosure));
                  Inv[Record].StandHeight                 =         (ushort)(floor(RealHeight + 0.5));
                  Inv[Record].Rev                         =                   (float)RealRev;
                  Inv[Record].BigTrees                    =         (ushort)(floor(RealBigTrees * BIGTREES_EXP));
```

```
                    Inv[Record].Vegcode                     =       VegCode;
                    Inv[Record].Harvest                     =       (float)Harvest * (float)ACREEQ;

                    //The incoming fuel loads are in LBS - convert to TONS but keep some precision by multiplying
by FUEL_LOAD_EXP
                    Inv[Record].Litter                      =       (ushort)(RealLitter/LoadFactor);
                    Inv[Record].Class25                     =       (ushort)(RealClass25/LoadFactor);
                    Inv[Record].Class1                      =       (ushort)(RealClass1/LoadFactor);
                    Inv[Record].Class3                      =       (ushort)(RealClass3/LoadFactor);
                    Inv[Record].Class6                      =       (ushort)(RealClass6/LoadFactor);
                    Inv[Record].Class12                     =       (ushort)(RealClass12/LoadFactor);
                    Inv[Record].ClassOver12                 =       (ushort)(RealClassOver12/LoadFactor);

                    //*************************************************************************************
                    //  Below is the Bernie "tweak" for CBD                         17Feb00
                    //*************************************************************************************
                    ModCbd = RealCBD * ( RealClosure * ACREEQ) ;

                    if( ModCbd > .30 )
                              ModCbd = .30;
                    else if(ModCbd < 0 )
                              ModCbd = 0;

                    //Store the kg per m3 data in the Inv[] structure
                    Inv[Record].Density = (ushort)(floor(ModCbd*DENSITY_EXP));

                    //*************************************************************************************
                    //  Call up the new Stand HLC function                          17Feb00
                    //  Use the method employed in StandDataController()
                    //*************************************************************************************
                    //Initialize StandClass and its pointer
                    ptr_stand = &StandClass;
                    memset(ptr_stand, 0, sizeof(struct STAND_CLASS) );

                    //Fill some items in ptr_stand-> before sending off      25Feb...why do this?
                    ptr_stand->Closure  = Inv[Record].Closure;

                    TempCode = Inv[Record].Vegcode;                                         //The actual 3 or digit
code from PREMO

                    //NOTE:  This TempCode Premo value DOES NOT have the modified MC vegclass to distinguish
between
                    //MC < 3000 and MC > 3000.  HOWEVER, this is OK for here because the NewStandHLC() function
will
                    //give the same HLC either way.

                    //extract the digits out
                    TempCover = TempCode%10;                                                        //last digit
for determining stage (is closure, <=60% or > 60% )
                    TempDiam = ( (TempCode-TempCover)%100 ) / 10;     //next to last digit also for determining
stage (is the QMD group)
                    TempVeg = (TempCode-TempCode%100) / 100;                    //lst  digit for determining
VegCode

                    //Put those values in ptr_stand->
                    ptr_stand->VegClass           = (ushort)TempVeg;
                    ptr_stand->Qmd                          = (ushort)TempDiam;
                    ptr_stand->CoverClass         = (ushort)TempCover;

                    //   NEW HLC stuff!!!!!
                    NewStandHLC(ptr_stand);
                    Inv[Record].HeightCrown       = (ushort)(floor(ptr_stand->HeightCrown + .5));


                    Record++;                                                               //BE SURE to increment
this counter up

          }//end for(y=0;y<NP;y++)

          fclose(DataIn);
}//end for(a = 0; a < Count; a++)

if(Record != RecordsNeeded)
          Bailout(96);


//Sort the current Inv structures by Treelist-Goal-Hold-Period
qsort(  (void*)Inv,
                    //base
        (size_t)RecordsNeeded,
                    //count of records
                    sizeof( struct PREMO_RECORD ),
                    //size of each record
                    LookAtPremoRecords );
                              //compare function


//Tell ptr_info where the memory is that holds this Inv[] stuff and how many records there are
ptr_info->CurrentSD = Inv;                                  //although Inv is an array of structures, by itself it
points to the memory block
ptr_info->Records   = RecordsNeeded;

//NOTE:  Don't delete the Inv[] structures here, they will be used in another function

}//end CreatePostDisturbanceStandDataStructure
```

```
//*****************************************************************************************
void MakeNewPremoRuns( struct TREELIST_FOR_PREMO FP[], int Count, int ActualPer)
//*****************************************************************************************
{
// Make Premo runs for those treelist hit by some type of disturbance this past period

FILE *OpenBatch;
char Batch[250], Parameters[250], JunkFile[256], RunPremo[256];

int a;

//For timing stuff
clock_t Start, Finish;
double Duration;

//---------------------------- End of variable defining -------------------------------------

/*
printf("\n\n============= The FP structures as MakeNewPremoRuns sees it ==================================\n");
for(a=0;a<Count;a++)
          printf("FP[%d]: Treelist: %lu\tGoal: %hu\tHold: %hu\tInterval: %hu\n",a,FP[a].Treelist,
                                                                        FP[a].Goal, FP[a].Hold,
FP[a].Interval);
*/

//Start the overall clock
Start = clock();

//Make the JunkFile which screen outputs can be redirected to
sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,ErrorDir);


//Go through the FP structures and make Premo calls for all needed records
for(a=0;a<Count;a++)                                                                    //for
each record in the FP structures
{

/*
Premo wants the following information passed in as a parameter for each call

1)        The full pathname of the Premo program itself
2)        The current period
3)        Treelist number
4)        Goal number
5)        Hold number
*/

          //Create a batch file to use to call up Premo
          sprintf(Batch, "%s%s%d\\per%d\\Premo.bat",PREFIX,OUTPUTS,GOAL_TO_USE,ActualPer);

          //Create the argument line that will get inputted into the above batch file
          sprintf(Parameters, "%s%s %d %lu %hu %hu %d",
                                                  PREFIX,PremoProgName,
                                                  ActualPer,
                                                  FP[a].NewTreelist,
                                                  FP[a].Goal,
                                                  FP[a].Hold,
                                                  PREMO_TOGGLE);


          //Open the batch file and write out the Parameters line and then close the file
          OpenBatch = fopen(Batch, "w" );
          if( OpenBatch == NULL)
                    fprintf(stderr, "Opening of %s failed: %s\n", Batch, strerror(errno) );
          fprintf(OpenBatch, "%s\n", Parameters);
          fclose(OpenBatch);

          //Fill the RunPremo array with the Batch file name and the screen output redirection file name
          sprintf(RunPremo, "%s > %s\n",Batch, JunkFile);

          //timing info local to this loop
          //clock_t Start, Finish;
          //double Duration;

          // Call up PREMO and run it with the above .bat file
          //Start = clock();
          system(RunPremo);
          //Finish = clock();
          //Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
          //printf("**That prescription took %.2lf seconds**\n", Duration );


}         //end of for(a=0...


Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("***All of those prescription took %.2lf seconds**\n", Duration );
printf("=============================================================================================\n\n");

for(a=0;a<10;a++)
          printf("\a");                                   //an alarm to tell it's finished
```

```
}//end MakeNewPremoRuns

//*********************************************************************************************
int FillForPremo( struct TREELIST_FOR_PREMO FP[], struct HIT_BY_DISTURB HitList[], int Count)
//*********************************************************************************************
{
//Go through HitList[] again and find those actual Unique combinations of Treelist-Goal-Hold counted earlier
//and this time fill up the FP structure
int a, b, Unique;
ulong EvalTreelist;
ushort EvalGoal, EvalHold;

//---------------------------- End of variable defining ----------------------------------------------
Unique = 0;
b = 0;                                                                                    //This must be
reset because above it left loop with value of Count
for(a=0;a<Count;)                                                          //a will get increment by other
loop
{
          if(b == Count)                                                                  //because of weird
incremental method, b will reach end first but a doesn't know that
                    break;

          Unique++;                                                                       //first one always counts
as do others because a gets reset in other loop

          //Set the initial Eval* variables
          EvalTreelist        = HitList[a].Treelist;
          EvalGoal            = HitList[a].Goal;
          EvalHold            = HitList[a].Hold;


          //Insert those values in the array of FP structures
          FP[Unique-1].NewTreelist      = EvalTreelist;
          FP[Unique-1].Goal                     = EvalGoal;
          FP[Unique-1].Hold                     = EvalHold;


          //sine HitList is already sorted, start at next record and look downward until no longer a match
          for(b=a+1;b<Count;)
          {
                    if(       HitList[b].Treelist == EvalTreelist &&
                              HitList[b].Goal             == EvalGoal &&
                              HitList[b].Hold             == EvalHold )

                              b++;
                                        //Then look at next record
                    else
                    {
                              //Set the "a" variable to where "b" is because this is the next unique match
                              a = b;
                              break;
                    }
          }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)


return Unique;

}//end FillForPremo


//*********************************************************************************************
int FillHitListForPremo( struct HIT_BY_DISTURB HitList[] , ulong FTL )
//*********************************************************************************************
{
/*
Use the same rules as in the CountTotalHit() function to determine whether or not a cell was
hit by ANY type of disturbance this period.

NOTE:  The array of HitList structures are type HIT_BY_DISTURB which is a structure type
developed to hold data after individual disturbances.  It became apparent that I could just
re-use this structure type to hold information for all the cells at the end of a period.  Just
be aware how certain members are being used.

Treelist: Normally would hold an OLD treelist value but in this case holds the updated and current treelist value

*/
int a, Count;

//----------------------- End of variable defining ----------------------------

//Fill up HitList
Count = 0;
for(a=0;a<UNIQUE;a++)
{
          if( Data.Cellid[a] == FALSE )
                    break;                                                                  //done looking through
arrays

          if( Data.Treelist[a] >= FTL )
          {
                    HitList[Count].Treelist       = Data.Treelist[a];
                    HitList[Count].Goal           = Data.Goal[a];
                    HitList[Count].Hold           = Data.Hold[a];
```

```
                    HitList[Count].Cellid         = Data.Cellid[a];

                    Count++;
          }
}

return Count;

}//end FillHitListForPremo



//*************************************************************************************************
int CountTotalHit(ulong FTL)
//*************************************************************************************************
{
/*
After all the disturbances in a period - the best way to see if a cell has been hit by ANY
type of disturbance is to look through the entire Data.*[] arrays and count how many have
a current Treelist value >= to the FirstTreelist value used in the period.

REMEMBER: this works because the Treelist value is updated right after each disturbance
*/
int a, Count=0;
//-------------------------------- End of variable defining ------------------------

for(a=0;a<UNIQUE;a++)
{
          if( Data.Cellid[a] == FALSE )
                    break;                                                          //done looking through
arrays

          if( Data.Treelist[a] >= FTL )
                    Count++;
}

return Count;
}//end CountTotalHit

//*************************************************************************************************
void InitialPremo(void)
//*************************************************************************************************
{

/* This function has TWO roles.....one is to create a RUNSTANDOPT.BAT file for EVERY stand (cell) that
needs to be optimized.  Second, once that .bat file is created this function will call up Heidi's program
(currently called Premo.exe) and pass the arguments from the .bat file to it.
*/


FILE *Index, *OpenBatch;
char TreeFileName[200]="", Garbage[200]="",Temp[256],Temp2[256],JunkFile[256];
int ScanStatus,IndexNo,count, ctr, goal, HoldPeriods;


/*
GOALS numbers as used in PREMO are:
0=Fire,             1=Insects,                              2=Fish,
          3=Wildlife-complex,           4=Wildlife-simple,
5=PNV,              6=Fire,Insects,PNV  7=Fish,Wildlife,PNV           8=All,PNV
          9=GrowOnly
10 = Non-forest(no goal associated) - only seen at end of simulation when outputting a goal value to map
*/

//For Time information
//clock_t Start, Finish;
//double Duration;

//------------------------------ End of variable defining ----------------------------------------

//First, delete the old prescriptions in the ....\initial\ directories
DeleteInitialStuff();

//Make the JunkFile which screen outputs can be redirected to
sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,ErrorDir);

// I will assume that the treelist index.txt file is completely filled with valid stands and files
sprintf(Temp, "%s%s%d\\per0\\%s",PREFIX,INPUTS,GOAL_TO_USE,TREE_INDEX);
Index = fopen(Temp,"r");

if (Index == NULL)
          fprintf(stderr, "opening of %s failed:  %s\n", Temp,  strerror(errno));


// First go through the file and COUNT the number of files
count = 0;
while ((ScanStatus=fscanf(Index,"%d",&IndexNo))!=EOF)
{
          count = ++count;
}

// Rewind the file pointer so it is back at the beginning of the file
rewind(Index);
```

```
// Now create a string that will be thrown into each runstandopt.bat file during the looping
// and then execute the .bat file for each stand
for(ctr = 0; ctr < count; ctr++)                                        //for each treelist
{
          fscanf(Index,"%d",&IndexNo);                                  //Scan the index no.

          printf("Working on Treelist %d\n",IndexNo);

          for(goal=0;goal<GOALS;goal++)                                         //for each goal
          {

                    //Set a quick error if I change the # of HoldFor periods and I forget to fix this code
                    if(HOLDNO > 2)
                              Bailout(40);

                    for(HoldPeriods=0;HoldPeriods<4;HoldPeriods+=3)   //for the two Hold "for" periods
                    {

                              //Instead of running PREMO for GrowOnly prescriptions with multiple "hold" values,
just
                              //copy the files from the hold==0 stuff into new files
                              if(goal == 9 && HoldPeriods > 0 )
                              {
                                        CopyStandOpt(IndexNo,goal,HoldPeriods);
                                        continue;                                                       //next
iteration of the for(HoldPeriods...) loop
                              }

                              sprintf(Temp, "%s%s 0 %d %d %d %d\n",PREFIX,PremoProgName, IndexNo, goal,
HoldPeriods, PREMO_TOGGLE);
                              sprintf(Temp2, "%s%s%d\\per0\\Premo.bat",PREFIX,INPUTS,GOAL_TO_USE);

                              //Open the PremoBatchFile and put in command line
                              OpenBatch = fopen(Temp2,"w");
                              if (OpenBatch == NULL)
                                        fprintf(stderr, "opening of %s.bat failed:  %s\n", Temp2 ,strerror(errno));
                              fprintf(OpenBatch, "%s\n", Temp);
                              fclose(OpenBatch);


                              //Refill Temp with the batch file name and the screen output redirection
                              sprintf(Temp, "%s > %s",Temp2,JunkFile);

                              // Call up PREMO and run it with the above .bat file
                              //Start = clock();
                              system(Temp);
                              //Finish = clock();
                              //Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
                              //printf("***That prescription took %.2lf seconds**\n", Duration );

                    }//end of for(HoldPeriods...
          }//end of for(goal....
}         //end of for(ctr...

          fclose(Index);

for(ctr=0;ctr<10;ctr++)
          printf("\a");                                         //an alarm

}//end of InitialStandOpt


//*********************************************************************************************
void CopyStandOpt(int Treelist, int Goal, int Hold)
//*********************************************************************************************
{
/*This function will copy over those PRESCRIPTION, VEGCODE, and STAND DATA files that were
generated for a Grow Only goal with a hold value of 0.  Any addition hold values are redundant
because hold refers to how many periods to "hold" before allowing cutting and in the Grow Only
there is no cutting - thus they will be the same.  At this time, I still need the files to
exist on the hard drive with the unique name.
*/
char CopyPres[300];
char CopyStand[300];
char JunkFile[300];

//Make the JunkFile which screen outputs can be redirected to
sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,ErrorDir);


if(Treelist == NONFOREST)                         //do nothing
          return;

if(Treelist < NONFOREST)                          //copy stuff from/to the INITIAL directory
{
          sprintf(CopyPres, "copy %s%s\\P_%d_%d_0.txt %s%s\\P_%d_%d_%d.txt >
%s",PREFIX,InitialPresDir,Treelist,Goal,

                                                            PREFIX,InitialPresDir,Treelist,Goal,Hold,JunkFile);
          sprintf(CopyStand, "copy %s%s\\SD_%d_%d_0.txt %s%s\\SD_%d_%d_%d.txt >
%s",PREFIX,InitialStandDataDir,Treelist,Goal,


PREFIX,InitialStandDataDir,Treelist,Goal,Hold,JunkFile);
}
```

```
else                                                                    //copy stuff from/to the MODELED
directory
{
          sprintf(CopyPres, "copy %s%s\\P_%d_%d_0.txt %s%s\\P_%d_%d_%d.txt >
%s",PREFIX,ModeledPresDir,Treelist,Goal,

                                                          PREFIX,ModeledPresDir,Treelist,Goal,Hold,JunkFile);

          sprintf(CopyStand, "copy %s%s\\SD_%d_%d_0.txt %s%s\\SD_%d_%d_%d.txt >
%s",PREFIX,ModeledStandDataDir,Treelist,Goal,


PREFIX,ModeledStandDataDir,Treelist,Goal,Hold,JunkFile);
}

//now execute those system calls
system(CopyPres);
system(CopyStand);

}//end CopyStandOpt



//*********************************************************************************************
void CreateTreeIndex(void)
//*********************************************************************************************
{
          // This function is to look at Data.Treelist and make a list of
          // all the unique treelist values that occur.  I will then take each of those unique values and
          // look for them in the Initialtreeindex.txt and create a new text file which contains ONLY those
          // index numbers for treelist values that were found in Data.Treelist

int b,c,next,there;
ulong *ptr_utl;
FILE *Master,*Current;
int ScanStatus,IndexNo,Found;
char TreeFileName[256],Temp[256],Temp2[256];
ulong *ptr_treelist;
int NumberTreelist = 0;
//------------------------------ End of variable defining ----------------------------------------

//First, count the total number of values in Data.Treelist
for(c=0;c<UNIQUE;c++)
{
          ptr_treelist = &Data.Treelist[c];
          if(*ptr_treelist != 0 && *ptr_treelist != NONFOREST)
                    NumberTreelist++;
}
//printf("\nThere are %d values (not necessarily unique) in Data.Treelist\n\n",NumberTreelist);


//Create and initialize the UniqueTreeList[] array on the free store
ulong (*UniqueTreeList) = new ulong[NumberTreelist];
if(UniqueTreeList == NULL)
          printf("Problems allocating memory for UniqueTreeList with %lu elements\n",NumberTreelist);

memset(UniqueTreeList, 0, sizeof(UniqueTreeList[0])*NumberTreelist);


//Then, pick out the unique numbers in Data.Treelist
next = 0;
for(c=0;c<UNIQUE;c++)
{
          there = 0;
          ptr_treelist=&Data.Treelist[c];

          if(*ptr_treelist == 0)
                    break;                                  //assumes Data.Treelist was initialized with 0's and
there are no treelist values of 0

          if(*ptr_treelist == NONFOREST)
                    continue;                                                   //don't put these in the file -
continue on to next

          for(b=0;b<NumberTreelist;b++)
          {
                    ptr_utl = &UniqueTreeList[b];           //utl stands for UniqueTreeList
                    if( (*ptr_treelist) == (*ptr_utl) )
                    {
                              there = 1;
                              break;                        //means the value is already in the UniqueTreeList[]
array
                    }
          }

          if(there == 0)
          {
                    UniqueTreeList[next] = *ptr_treelist;
                    next++;
          }

}


//Now create a TREEINDEX.txt file to be placed in the /inputs/per0 directory so that the PremoStuff.cpp
```

```
// functions can open that file and run Premo for these initial treelist.
          sprintf(Temp, "%s%s\\%s",PREFIX,ConstantInput,IT_INDEX);
          sprintf(Temp2, "%s%s%d\\per0\\%s", PREFIX,INPUTS,GOAL_TO_USE,TREE_INDEX);

          Master = fopen(Temp,"r");
          Current = fopen(Temp2,"w");

          if (Master == NULL || Current == NULL)
                    fprintf(stderr, "Opening of %s or %s failed:  %s\n",Temp,Temp2,strerror(errno));


//Now look through the UniqueTreeList[] array and for each of the values in it, find the file location
//from the InitialTreeindex.txt and send both that file directory and the unique tree list value to a new
//text file called Treeindex.txt (placed in the /inputs/per0/ directory

          for(c=0;c<next;c++)
          {

                    Found = 0;
                    while ((ScanStatus=fscanf(Master,"%d %s",&IndexNo,TreeFileName))!=EOF)
                    {
                              if (IndexNo == (int)UniqueTreeList[c])
                              {
                                        fprintf(Current,"%d\n",IndexNo);

                                        Found = 1;
                                        rewind(Master);
                                        break;
                              }
                    }//end of while

                    if (Found == 0)
                    {
                              printf("There appears to be no treelist available for IndexNo: %d - Bailing
out!\n",UniqueTreeList[c]);
                              Bailout(20);
                    }

          }//end of for(c=0;c<next;c++)

          fclose(Master);
          fclose(Current);

//delete stuff on free store
delete [] UniqueTreeList;

//Set the global variable ITL so the value of next
ITL = next;

} //end of CreateTreeIndex function


//*****************************************************************************************
void CreateSortedPremoBinaryFile(void)
//*****************************************************************************************
{
/*
This function will create a sorted binary file that contains all the SD_*_*_*.txt
data for the current landscape.  This file only needs to be created when PREMO has been
ran on the initial files (i.e. after a change in PREMO coding).  The resulting binary file
can then be read in later during FillInitialPremoData().

*/

FILE *Index, *DataIn, *BinOut, *HeaderOut;
char Temp[256];
int ScanStatus,Treelist,count, NonForestCount, RealCount, ctr, goal, HoldPeriods;
int y;
ulong RecordsNeeded, Record;
int DataPeriod;
double RealBasal, RealClosure, RealCBD, RealHLC, RealHeight, RealReV, RealBigTrees, Harvest;
ushort VegCode;
double RealLitter, RealClass25, RealClass1, RealClass3, RealClass6, RealClass12, RealClassOver12;
double LoadFactor;
//------------------------------ End of variable defining ---------------------------------------------

//Figure out how many records there are going to be
//I will assume that the treelist index.txt file is completely filled with valid stands and files
sprintf(Temp, "%s%s%d\\per0\\%s",PREFIX,INPUTS,GOAL_TO_USE,TREE_INDEX);
Index = fopen(Temp,"r");
if (Index == NULL)
          fprintf(stderr, "opening of %s failed:  %s\n", Temp, strerror(errno));

// First go through the file and COUNT the number of files
count = 0;
NonForestCount=0;
while ((ScanStatus=fscanf(Index,"%d",&Treelist))!=EOF)
{
          if( Treelist == NONFOREST)
                    NonForestCount++;

          count = ++count;
}

RealCount = count - NonForestCount;
```

```
//printf("There are %d files in the treeindex.txt file but only %d are Forest type\n",count,RealCount);

// Rewind the file pointer so it is back at the beginning of the file
rewind(Index);

//Determine the actual # of records
RecordsNeeded = RealCount * GOALS * HOLDNO * NP;            //Use RealCount because records for NONFOREST are not
needed


//Create an array of structures on the free store to hold these
struct PREMO_RECORD (*Inv) = new struct PREMO_RECORD[RecordsNeeded];
if( Inv == NULL )
         printf("Problems allocating memory for Inv[] with %lu elements\n",RecordsNeeded*sizeof(PREMO_RECORD));
//Initialize
memset( Inv, 0, sizeof(struct PREMO_RECORD) * RecordsNeeded );



//*****************  Start pumping data into the array of Inv[] structures **********************

//Set the LoadFactor that will be used to convert the incoming FuelLoadings (which are in LBS) to TONS but
//keeping some precision by multiplying by FUEL_LOAD_EXP
LoadFactor          = TONS * FUEL_LOAD_EXP;

Record=0;

for(ctr=0;ctr<count;ctr++)                                                          //for all the treelist
listed in the treeindex.txt file
{
         fscanf(Index,"%d",&Treelist);                                          //Scan the actual treelist number


         if(Treelist == NONFOREST)                                                  //no need to do these
                  continue;

         for(goal=0;goal<GOALS;goal++)                                          //for each goal
         {

                  //Set a quick error if I change the # of HoldFor periods and I forget to fix this code
                  if(HOLDNO > 2)
                           Bailout(40);

                  for(HoldPeriods=0;HoldPeriods<4;HoldPeriods+=3)   //for the two Hold "for" periods
                  {

                           //Start by opening the correct SD_*_*_*.txt file ONCE
                           sprintf(Temp,
"%s%s\\SD_%d_%d_%d.txt",PREFIX,InitialStandDataDir,Treelist,goal,HoldPeriods);
                           DataIn = fopen(Temp, "r");
                           if (DataIn == NULL)
                                    fprintf(stderr, "opening of %s failed: %s\n",Temp, strerror(errno));



                           //The SD_*_*_*.txt files will have a line of data for the incoming treelist at and it
is NOT needed.
                           //SafeD works by assuming the the 1st period is after Harvest activities and 5-yr
growth
                           fscanf(DataIn,"%d %lf %lf %lf %lf %lf %lf %lf %hu %lf %lf %lf %lf %lf %lf %lf %lf",
&DataPeriod, &RealBasal,
                                    &RealClosure, &RealCBD, &RealHLC, &RealHeight, &RealRev, &RealBigTrees,
&VegCode, &Harvest,
                                    &RealLitter, &RealClass25, &RealClass1, &RealClass3, &RealClass6,
&RealClass12, &RealClassOver12);


                           //Now scan in all the data ONCE and store in the array of Inv structures
                           //I assume that the SD_*_*_* have Period 1 on line 2
                           for(y=0;y<NP;y++)
                           {
                                    //First, scan in the lines from SD*
                                    fscanf(DataIn,"%d %lf %lf %lf %lf %lf %lf %lf %hu %lf %lf %lf %lf %lf %lf
%lf %lf",   &DataPeriod, &RealBasal,
                                             &RealClosure, &RealCBD, &RealHLC, &RealHeight, &RealRev,
&RealBigTrees, &VegCode, &Harvest,
                                             &RealLitter, &RealClass25, &RealClass1, &RealClass3, &RealClass6,
&RealClass12, &RealClassOver12);

                                    if(DataPeriod != y+1)
                                             printf("problem with period values in the PREMO files\n");

                                    //Check and make sure values that eventually will be converted to USHORT
are OK in terms of 65535 thingy!!
                                    if(RealBasal >= 6553  || RealCBD >= 655 || RealBigTrees >= 6553 ||
RealLitter >= 6553*TONS ||
                                             RealClass25 >= 6553*TONS || RealClass1 >= 6553*TONS || RealClass3
>= 6553*TONS ||
                                             RealClass6 >= 6553*TONS || RealClass12 >= 6553*TONS ||
RealClassOver12 >= 6553*TONS)
                                    {
                                             printf("Guilty is:  Tree %d  Goal %d and Hold %d in period %d\n",
Treelist, goal, HoldPeriods,y+1);
                                             Bailout(49);
```

```
                                        }

                                        //Just in case
                                        if(RealCBD < 0 )
                                                RealCBD = 0;

                                        Inv[Record].Treelist                    =       (ulong)Treelist;
                                        Inv[Record].Goal                        =       (ushort)goal;
                                        Inv[Record].Hold                        =       (ushort)HoldPeriods;
                                        Inv[Record].Period                      =       (ushort)y;
                                        Inv[Record].Basal                       =       (ushort)(floor(RealBasal
* BASAL_EXP));
                                        Inv[Record].Closure                     =
        (ushort)(floor(RealClosure));
                                        Inv[Record].Density                     =       (ushort)(floor(RealCBD *
DENSITY_EXP));
                                        Inv[Record].HeightCrown                 =       (ushort)(floor(RealHLC +
0.5));
                                        Inv[Record].StandHeight                 =       (ushort)(floor(RealHeight
+ 0.5));
                                        Inv[Record].Rev                                 =       (float)RealRev;
                                        Inv[Record].BigTrees                    =
        (ushort)(floor(RealBigTrees * BIGTREES_EXP));
                                        Inv[Record].Vegcode                     =       VegCode;
                                        Inv[Record].Harvest                     =       (float)Harvest;

                                        //The incoming fuel loads are in LBS - convert to TONS but keep some
precision by multiplying by FUEL_LOAD_EXP
                                        Inv[Record].Litter                      =
        (ushort)(RealLitter/LoadFactor);
                                        Inv[Record].Class25                     =
        (ushort)(RealClass25/LoadFactor);
                                        Inv[Record].Class1                      =
        (ushort)(RealClass1/LoadFactor);
                                        Inv[Record].Class3                      =
        (ushort)(RealClass3/LoadFactor);
                                        Inv[Record].Class6                      =
        (ushort)(RealClass6/LoadFactor);
                                        Inv[Record].Class12                     =
        (ushort)(RealClass12/LoadFactor);
                                        Inv[Record].ClassOver12                 =
        (ushort)(RealClassOver12/LoadFactor);


                                        Record++;                                                       //BE
SURE to increment this counter up

                                }//end for(y=0;y<NP;y++)

                                fclose(DataIn);
                        }//end for(HoldPeriods=0
                }//end for(goal=0;goal<GOALS;goal++)
}//end for(ctr = 0; ctr < count; ctr++)

fclose(Index);


//***** Sort the array by Treelist-Goal-Hold-Elev-Period and prepare to write out the results
qsort(  (void*)Inv,
                        //base
        (size_t)RecordsNeeded,
                //count of records
                sizeof( struct PREMO_RECORD ),
                //size of each record
                LookAtPremoRecords );
                        //compare function


//Create the output Binary file and header file
sprintf(Temp, "%s%s\\Binary\\%s_Premo.bin",PREFIX,InitialStandDataDir,ENVT);
BinOut = fopen(Temp, "wb");

sprintf(Temp, "%s%s\\Binary\\%s_Premo.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderOut = fopen(Temp, "w");

//Write out the header data -- need to know how many records there are
fprintf(HeaderOut, "%lu\n",RecordsNeeded);

//And now write out all the records in the array of Inv structure
fwrite(Inv, sizeof(PREMO_RECORD),RecordsNeeded,BinOut);

fclose(BinOut);
fclose(HeaderOut);

delete [] Inv;
}//end CreateSortedPremoBinaryFile

//*************************************************************************************************
int LookAtPremoRecords(const void *ptr1, const void *ptr2)
//*************************************************************************************************
{

        //Just to typecast them since we aren't actually passing in pointers
        struct PREMO_RECORD *elem1;
        struct PREMO_RECORD *elem2;
```

```
        elem1 = (struct PREMO_RECORD *)ptr1;
        elem2 = (struct PREMO_RECORD *)ptr2;


    if( elem1->Treelist < elem2->Treelist )
          //First sort by Treelist
        return -1;
    if( elem1->Treelist > elem2->Treelist )
        return 1;
    else
                                    //Then by Goal
          {
                    if( elem1->Goal < elem2->Goal )

                              return -1;
                    if( elem1->Goal > elem2->Goal )
                              return 1;
                    else
                                        //Then by Hold
                    {
                              if( elem1->Hold < elem2->Hold )

                                        return -1;
                              if( elem1->Hold > elem2->Hold )
                                        return 1;
                              else
                                        //Then by Period

                              {
                                        if( elem1->Period < elem2->Period )

                                                  return -1;
                                        if( elem1->Period > elem2->Period )
                                                  return 1;
                                        else
                                                  return 0;
                                        //FINISHED!!
                              }//end Period
                    }//end Hold
          }//end Goal

}//end LookAtPremoRecord

//*****************************************************************************************
void FillInitialPremoData(int per)
//*****************************************************************************************
{
/*
This function will be called up only at period 0, which is before main()
starts into the NP period loop.  This function needs to COMPLETELY fill up the
Data.* array for all NP periods with the initial data derived from the prescription generator.

Filling in the initial data will be done by using the sorted binary file created during
CreateSortedPremoBinaryFile()
*/

//IO variables
FILE *BinIn, *HeaderIn;
char Temp[256];

//structures
struct PREMO_RECORD Key;
struct PREMO_RECORD *ptr_record;

//pointers
ulong *ptr_treelist;
ushort *ptr_goal,  *ptr_hold, *ptr_elev, *ptr_vegcode;


int x,y, count;
ulong RecordNo;


//For Time information
clock_t Start, Finish;
double Duration;

// --------------------------------End of variable defining ----------------------------------------------------

Start = clock();

printf("\n\n=============================================================================================================
========\n");
printf("\t\t***  Filling the Data.* arrays with Binary data from PREMO ***\n");

//Create and Open the Header and actual Binary file with PREMO data in it
sprintf(Temp, "%s%s\\Binary\\%s_Premo.bin",PREFIX,InitialStandDataDir,ENVT);
BinIn = fopen(Temp, "rb");

sprintf(Temp, "%s%s\\Binary\\%s_Premo.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderIn = fopen(Temp, "r");


//Get the Number of records that are listed in the header file
```

```
fscanf(HeaderIn,"%lu",&RecordNo);

//Create an array of structures on the free store to hold these records
struct PREMO_RECORD (*PremoInv) = new struct PREMO_RECORD[RecordNo];
if( PremoInv == NULL )
        printf("Problems allocating memory for PremoInv[] with %lu elements\n",RecordNo*sizeof(PREMO_RECORD));


//Initialize a couple of things
memset( PremoInv, 0, sizeof(struct PREMO_RECORD) * RecordNo );
memset( &Key, 0, sizeof( struct PREMO_RECORD) );


//Now just read in the binary data the same way it was written out in CreateSortedPremoBinaryFile()
fread(PremoInv, sizeof(PREMO_RECORD),RecordNo,BinIn);

//close up the files
fclose(BinIn);
fclose(HeaderIn);


//Go one-by-one through the Data.* stands and get the Treelist-goal-hold-Period and then use that as
//a key to do a binary search on the PremoInv structures and find the correct data
//I assume that all Data.* arrays are filled and that the first Data.Treelist element
//has a value in it - if I hit a Data.Treelist == 0 then there are no more to do on landscape
for(count=0;count<UNIQUE;count++)                        //go one-by-one through Data.Treelist[]
{

//Set the primary pointers to the Data.* arrays
ptr_treelist =      &Data.Treelist[count];
ptr_goal =                  &Data.Goal[count];
ptr_hold =                  &Data.Hold[count];
ptr_elev =                  &Data.Elev[count];
ptr_vegcode =       &Data.Vegcode[count][0];


        //Set a break if we get a treelist == 0 ...signals the end of data in the Data.*[] arrays
        if( *ptr_treelist == FALSE )
                break;

        //If this cell is NONFOREST then make some adjustments to fuel models and fill Data.Vegcode[][] with
NONFOREST
        if(*ptr_treelist == NONFOREST)
        {
                //NOTE: the NONFOREST Data.InitialFuelModel[] and Data.FuelModel[][] were filled up back in the
function
                //LoadInitialFuelModelsAndLoads().  I am not going to double-check here that it was done
although I have
                //ran test to make sure.

                //Assign a goal of "10" to indicate this is GROW-ONLY (non-forest)
                *ptr_goal = 10;

                //Set all Data.Vegcode values to NONFOREST
                for(x=0;x<NP;x++)
                {
                        *ptr_vegcode = NONFOREST;
                        ptr_vegcode++;
                }

                continue;                                       //should continue next iteration of big
for(count=0;count<UNIQUE...) loop
        }//end if(*ptr_treelist == NONFOREST)

//====================== End of what to do if the initial treelist is NONFOREST  ======================

        else if( *ptr_vegcode == 0)                     //This treelist-goal combination has NOT been inputted
yet
        {                                                                                               //All the
Vegcodes should be 0 at beginning of simulation

                //Start to make some of the "Key" for this cell to use in looking for the correct
                //record in the array of PremoInv structures
                Key.Treelist                    = (ulong)*ptr_treelist;
                Key.Goal                        = (ushort)*ptr_goal;
                Key.Hold                        = (ushort)*ptr_hold;


                //make another loop to account for the period
                for(y=0;y<NP;y++)
                {
                        Key.Period = (ushort)y;

                        //printf("Key: %lu %hu %hu %hu %hu\n",Key.Treelist,Key.Goal,Key.Hold,Key.Period);

                        //Now use bsearch to find the matching record in the array of PremoInv structures
                        ptr_record = (struct PREMO_RECORD*)bsearch(
                                &Key,
                                (void *)PremoInv,
                                (size_t)RecordNo,
                                sizeof( struct PREMO_RECORD),
                                LookAtPremoRecords  );

                        if(ptr_record == NULL)
                                Bailout(75);
```

```
                                //Check to see if the Vegcode value is Mixed Conifer - if so it must be broken into >
or < 3000'
                                if( (int)(ptr_record->Vegcode / 100 ) == 5 )
                                {
                                        if( *ptr_elev >= (3000*FT2M) )
                                                                //It's over 3,000 ft
                                                Data.Vegcode[count][y]                  =       ptr_record-
>Vegcode + 500;                 //This will give it 10**
                                        else
                                                Data.Vegcode[count][y]                  =       ptr_record-
>Vegcode;                       //Leave as is
                                }
                                else
                                        Data.Vegcode[count][y]                          =       ptr_record-
>Vegcode;                       //Leave as is

                                if(Data.Vegcode[count][y] > 1061 )
                                        printf("Got a BAD vegcode value during fill data with initial PREMO
data\n");


                                //Fill in the rest of Data.*[] arrays with the data accessible from the pointer
returned above
                                //Everyone should already have the proper type - converted back in
CreateSortedPremoBinaryFile()
                                Data.Basal[count][y]                    =       ptr_record->Basal;
                                Data.Closure[count][y]                  =       ptr_record->Closure;
                                Data.CBDensity[count][y]        =       ptr_record->Density;
                                Data.HLC[count][y]                      =       ptr_record->HeightCrown;
                                Data.StandHeight[count][y]      =       ptr_record->StandHeight;
                                Data.BigTrees[count][y]                 =       ptr_record->BigTrees;
                                Data.CFHarvest[count][y]        =       ptr_record->Harvest;

                                //Fuel Loads were also properly converted back in CreateSortedPremoBinaryFile()
                                Data.Litter[count][y]                                   =       ptr_record->Litter;
                                Data.Class25[count][y]                                  =       ptr_record->Class25;
                                Data.Class1[count][y]                                   =       ptr_record->Class1;
                                Data.Class3[count][y]                                   =       ptr_record->Class3;
                                Data.Class6[count][y]                                   =       ptr_record->Class6;
                                Data.Class12[count][y]                                  =       ptr_record->Class12;
                                Data.ClassOver12[count][y]              =       ptr_record->ClassOver12;


                }//end for(y=0;y<NP;y++)
        }//end of if( *ptr_vegcode == 0)
}//end for(count=0;count<UNIQUE;count++)

//delete any arrays on free store
delete [] PremoInv;


Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\t\tTo fill the Data.* arrays with PREMO stand data took %.2lf seconds\n", Duration );
printf("=============================================================================================================
====\n");
}//end of FillPremoData



//*****************************************************************************************
int CompareHitListForNewPremo(const void *ptr1, const void *ptr2)
//*****************************************************************************************
{

        //Just to typecast them since we aren't actually passing in pointers
        struct HIT_BY_DISTURB *elem1;
        struct HIT_BY_DISTURB *elem2;

        elem1 = (struct HIT_BY_DISTURB *)ptr1;
        elem2 = (struct HIT_BY_DISTURB *)ptr2;


    if( elem1->Treelist < elem2->Treelist )
          //First sort by Treelist
        return -1;
    if( elem1->Treelist > elem2->Treelist )
        return 1;
    else
                                //Then by Goal
        {
                if( elem1->Goal < elem2->Goal )

                        return -1;
                if( elem1->Goal > elem2->Goal )
                        return 1;
                else
                                        //Then by Hold
                {
                        if( elem1->Hold < elem2->Hold )

                                return -1;
                        if( elem1->Hold > elem2->Hold )
                                return 1;
```

```
                    else


                        return 0;
                    //FINISHED!!
                }//end Hold
        }//end Goal

}//end CompareHitListForNewPremo

//*************************************************************************************************
int CountUniqueNewPremoHits(struct HIT_BY_DISTURB HitList[], int Count)
//*************************************************************************************************
{
//Go through HitList[] and find how many actual Unique combinations of Treelist-Goal-Hold

int a,b,Unique;
ulong EvalTreelist;
ushort EvalGoal, EvalHold;
//------------------------ end of variable defining ---------------------------------------

Unique = 0;
b = 0;
for(a=0;a<Count;)                                                    //a will get increment by other
loop
{
        if(b == Count)                                                          //because of weird
incremental method, b will reach end first but a doesn't know that
                break;

        Unique++;                                                               //first one always counts
as do others because a gets reset in other loop

        //Set the initial Eval* variables
        EvalTreelist        = HitList[a].Treelist;
        EvalGoal            = HitList[a].Goal;
        EvalHold            = HitList[a].Hold;


        //sine HitList is already sorted, start at next record and look downward until no longer a match
        for(b=a+1;b<Count;)
        {
                if(     HitList[b].Treelist == EvalTreelist &&
                        HitList[b].Goal             == EvalGoal &&
                        HitList[b].Hold             == EvalHold
                        )

                        b++;
        //look at next record
                else
                {
                        //Set the "a" variable to where "b" is because this is the next unique match
                        a = b;
                        break;
                }
        }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)

return Unique;
}//end CountUniqueNewPremoHits


READDATA.CPP

//  This READDATA.cpp file is to hold the functions used to populate the DATA structure.
//  This structure has arrays that hold a bunch of input data themes.

//  There is also a link[][] array created that is the array used to "link" any input data into the
//  the arrays in DATA by means of a row/column search and linkage.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "globals.h"



//OK, see if I can make a structure called "Data" to store the maindata.
//Using a structure to keep memory usage down because only the Cellid and
//Treelist needs to be ulong(4bytes each element) and the others can be ushort.
//I could just declare each one separately but I want some practice with structures!

struct Main{
        ulong Cellid[UNIQUE];                                           //      Values for entire
Applegate watershed
        ushort GridRow[UNIQUE];
        ushort GridColumn[UNIQUE];
        ulong Treelist[UNIQUE];
        ushort Elev[UNIQUE];                                            //      in meters
        ushort Aspect[UNIQUE];
        ushort Slope[UNIQUE];
        ushort Goal[UNIQUE];
        ushort Owner[UNIQUE];
```

```
          ushort Pag[UNIQUE];
          ushort Alloc[UNIQUE];                                          //        Does NOT include an
allocation for stream buffers - use Data.Buffer
          ushort Minor[UNIQUE];
          ushort Hold[UNIQUE];
          ushort Buffer[UNIQUE];                                         //  Stream buffers on FED land
only!!  NODATAFLAG = noBuff, 100 = in Buffer
          ushort FireHistory[UNIQUE];                                //        Old fire perimeters
NODATAFLAG = not in, 100 = in old polygon
          ushort InitialVeg[UNIQUE];
          ushort InitialStage[UNIQUE];
          ushort PRule[UNIQUE];


          //Get calculated within SafeD
          ushort InitialDuff[UNIQUE];                                    // These initial Fuel Loadings will be
divided by TONS - convert back by
          ushort InitialLitter[UNIQUE];                          // multiplying by TONS when using.(also using
FUEL_LOAD_EXP to make ushort)
          ushort InitialClass25[UNIQUE];
          ushort InitialClass1[UNIQUE];
          ushort InitialClass3[UNIQUE];
          ushort InitialClass6[UNIQUE];
          ushort InitialClass12[UNIQUE];
          ushort InitialClassOver12[UNIQUE];
          ushort InitialFuelModel[UNIQUE];
          ushort InitialEra[UNIQUE];
          ushort FuelModel[UNIQUE][NP];
          ushort Duff[UNIQUE][NP];                                       //Divided by TONS - and multiply by
FUEL_LOAD_EXP
          ushort Flame[UNIQUE];                                                  //Will hold the current flame
length interval from a FARSITE run

          //Data that will come from Premo
          ushort Basal[UNIQUE][NP];                                      // converting...divide by 10 to get REAL
value
          ushort Closure[UNIQUE][NP];                                    // Truncating to closest integer
          ushort CBDensity[UNIQUE][NP];                          // converting...divide by 100 to get REAL value
          ushort HLC[UNIQUE][NP];                                                // Truncating to closest integer
          ushort StandHeight[UNIQUE][NP];                                // Truncating to closest integer
          ushort BigTrees[UNIQUE][NP];                           // converting...divide by 10 to get REAL value
          ushort Era[UNIQUE][NP];                                                // converting...divide by 100 to
get REAL value
          ushort Vegcode[UNIQUE][NP];                                    //
          float CPHarvest[UNIQUE][NP];                           //
          ushort    Litter[UNIQUE][NP];                                  //-------  All these fuel loadings will be
divided (/) by TONS -- Premo * by TONS
          ushort    Class25[UNIQUE][NP];                                 //-------  when it outputted and some are too
big to fit in ushort.  Convert
          ushort    Class1[UNIQUE][NP];                                  //-------  when needed back to TONS.
          ushort    Class3[UNIQUE][NP];
          ushort    Class6[UNIQUE][NP];
          ushort    Class12[UNIQUE][NP];
          ushort    ClassOver12[UNIQUE][NP];
          } Data;

//Array to hold the "linkage" for filling up the arrays in the Data structure
int link[ROWS][3];


//functions used in ReadData.cpp
int CreateMainData(void);
int AsciiReadCell(void);                                         //For Ascii reading
//int AsciiReadData(void);
int MakeLink(void);                                                      //both binary and ascii ways can use this
int BinaryReadData(void);                                        //For Binary reading
long CheckHeader(int File);
void ReadGoalHoldFound(int Goal);

//defined in StandOptStuff.cpp
extern void CreateTreeIndex(void);


//defined in Misc.cpp
extern void Bailout(int ErrorNumber);
extern void PrintToStat(int, ulong Value);

// ************************************************************
int CreateMainData(void)
// ************************************************************
{
#ifdef USE_BEST_GOAL_HOLD
          printf("\n\n******** Creating and initializing the main Data.*[] arrays..... **********\n");
          printf("*********   Will be using GOAL and HOLD values from previous simulation.  ************\n\n");
#else
          printf("\n\n******** Creating and initializing the main Data.*[] arrays..... **********\n");
#endif


int r,s;
//For Time information
clock_t Start, Finish;
double Duration;
```

```
//Initialize all the arrays of the Data.* arrays w/ 0's or other data
for(r=0;r<UNIQUE;r++)
{
          Data.Cellid[r]                                    = 0;
          Data.GridRow[r]                                   = 0;
          Data.GridColumn[r]                        = 0;
          Data.Treelist[r]                          = 0;
          Data.Elev[r]                                      = 0;
          Data.Aspect[r]                                    = 0;
          Data.Slope[r]                                     = 0;
          Data.Owner[r]                                     = 0;
          Data.Pag[r]                                                 = 0;
          Data.Alloc[r]                                     = 0;
          Data.Minor[r]                                     = 0;
          Data.Buffer[r]                                    = 0;
          Data.FireHistory[r]                       = 0;
          Data.InitialVeg[r]                        = 0;
          Data.InitialStage[r]                      = 0;
          Data.Flame[r]                                     = 0;
          Data.Goal[r]                                      = 9;                //Goal will get a "real" value in
PickPrescription - defaults to 9 (GrowOnly)
          Data.Hold[r]                                      = 0;                //same with Hold - defaults to 0
(available to cut in period 1)

          Data.InitialDuff[r]                       = 0;
          Data.InitialLitter[r]                     = 0;
          Data.InitialClass25[r]                    = 0;
          Data.InitialClass1[r]                     = 0;
          Data.InitialClass3[r]                     = 0;
          Data.InitialClass6[r]                     = 0;
          Data.InitialClass12[r]                    = 0;
          Data.InitialClassOver12[r]  = 0;
          Data.InitialFuelModel[r]      = 0;
          Data.InitialEra[r]                        = 0;
          Data.PRule[r]                                     = 0;

          for(s=0;s<NP;s++)                                                   //These all will get filled after
prescription is selected
          {
                    Data.Vegcode[r][s]              = 0;
                    Data.CFHarvest[r][s]            = 0;
                    Data.Basal[r][s]                = 0;
                    Data.Closure[r][s]              = 0;                //remember: the SD_*_*_*.txt file DOES have
canopy closure
                    Data.FuelModel[r][s]            = 0;
                    Data.CBDensity[r][s]            = 0;
                    Data.HLC[r][s]                            = 0;
                    Data.StandHeight[r][s]          = 0;
                    Data.BigTrees[r][s]             = 0;
                    Data.Era[r][s]                            = 0;
                    Data.Duff[r][s]                           = 0;
                    Data.Litter[r][s]               = 0;
                    Data.Class25[r][s]              = 0;
                    Data.Class1[r][s]               = 0;
                    Data.Class3[r][s]               = 0;
                    Data.Class6[r][s]               = 0;
                    Data.Class12[r][s]              = 0;
                    Data.ClassOver12[r][s]          = 0;
          }
}
printf("--------Finished initializing, now to start filling in with data----------\n");


Start = clock();

//NEW: Nov 99.  Call up functions to read either ASCII or BINARY files (use #define FILE_TYPE toggle)

#if (FILE_TYPE == 1)
                                                                                          //Read in the ASCII files

puts("\n\n************************ WARNING ***** WARNING ***** WARNING ***************************");
puts("*****                                                                    *****");
puts("*****   ASCII reading method has not been completely updated yet - will not input *****");
puts("***** the Slope, Aspect and Pag files - BAILING NOW! and see AsciiReadData()      *****");
puts("*****                                                                    *****");
puts("*******************************************************************************");

exit(1);

          //call up these routines to fill the Data.*[] and link[][] arrays
          if( AsciiReadCell() )
          {         if(MakeLink() )
                    {         //if( AsciiReadData() )
                              {
                                        Finish = clock();
                                        Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
                                        printf("\n**Reading the initial data in ASCII format took %.2lf
seconds**\n", Duration );
                              }
                    }
          }
```

```
#else
                                                                                    //Read in the
BINARY files
          //call up these routines to fill the Data.*[] and link[][] array
          if( BinaryReadData() )
          {
                    Finish = clock();
                    Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
                    printf("\n**Reading the initial data in BINARY format took %.2lf seconds**\n", Duration );

          }


#endif    //if(FILE_TYPE==1)


//***************** TREEINDEX Creation *************************
//Create the TreeIndex.txt file now, because too many other things rely on it
#ifdef CREATE_TREE_INDEX
CreateTreeIndex();                    //undefine & create new file with just 1 or 2 values when running test on
PREMO
#endif
//*************************************************************


return TRUE;

}//end of CreateMainData

// ***************************************************************************
// ***************************************************************************
//                                    Start of functions to handle importing the data in BINARY format
// ***************************************************************************
// ***************************************************************************
// ***********************************************************
int BinaryReadData(void)
// ***********************************************************
{

FILE *BIN;
char InFile[150]="";
int r,c;
ulong cell;
long CellidND;                                                                  //hold the returned NoData value
from CheckHeader() for Cellid - it is reused
long ND;                                                                        //hold other returned NoData values
from CheckHeader()
int FileNo;
float RawValue;



//Create a temporary array to store the input Cellid binary data, which has data for every cell
          float (*TempCellid)[COLUMNS] = new float[ROWS][COLUMNS];   //ROWS*COLUMNS is how many elements are in the
initial grid/binary file
          if (TempCellid == NULL)
                    printf("There was NOT enough memory for TempCellid with %lu elements\n",ROWS*COLUMNS);

//Initialize the TempCellid array
for(r=0;r<ROWS;r++)
{
          for(c=0;c<COLUMNS;c++)
                    TempCellid[r][c] = 0;
}

//Check the header data associated with this binary file and get the returned NODATA value
CellidND = CheckHeader(0);

//******read in every element of the Cellid data and store in the TempCellid array
sprintf(InFile, "%s%s\\cellid_%s.bin",PREFIX,ConstantInput,ENVT);
BIN = fopen(InFile, "rb");
if( fread(TempCellid,sizeof(TempCellid),ROWS*COLUMNS,BIN) != ROWS*COLUMNS)      //TempCellid is only a pointer!!
          Bailout(66);
else
          printf("**Binary file %s OK**\n",InFile);
fclose(BIN);



//Using the same criteria as reading in the old ASCII files, (i.e. checking for NODATA) fill the actual
//Data.Cellid[] array with only those values needing tracked and fill Data.GridRow[] with
//the original ROW cell number and the Data.GridColumn[] with original COLUMN cell.
cell = 0;
for(r=1;r<=ROWS;r++)
{
          for(c=1;c<=COLUMNS;c++)
          {
                    if(TempCellid[r-1][c-1] != CellidND)
                    {
                              Data.Cellid[cell] = (ulong)TempCellid[r-1][c-1];            //The CELLID value
                              Data.GridRow[cell] = (ushort)r;
                    //original grid row
                              Data.GridColumn[cell] = (ushort)c;
          //original column row
                              cell++;
```

```
                }
        }
}


//Call up the MakeLink() to create the link[][] array used throughout this program
if( MakeLink() == FALSE)
        Bailout(67);



/*
Loop through and read in the Input Landscape files.  This is being done one file at a time so
only one TempInput has to be created.  The TempCellid[][] created above will act as the
template for reading in the current landscape file.
*/


//Create a temporary array to store the input binary data, which has data for every cell
//NOTE: These input binary files are generated by ArcInfo as FLOATing numbers so use that as type and typecast
later as needed
float (*TempInput)[COLUMNS] = new float[ROWS][COLUMNS];   //ROWS*COLUMNS is how many elements are in the initial
binary file
if (TempInput == NULL)
        printf("There was NOT enough memory for TempInput with %lu elements\n",ROWS*COLUMNS);

for(FileNo=1;FileNo<=FILES;FileNo++)
{

/*I am going to hard-wire codes to use for REQUIRED input landscape files:
1       Treelist
2       Elevation
3       Aspect
4       Slope
5       Ownership
6       Plant Association Group
7       Allocation (for federal lands)
8       Minor (sub-watersheds)
9       Buffer (stream buffers on federal lands)
10      Fire History
11      InitialVeg
12      InitialStage
13  Prescription Rule allocation for Framework only

There will be Switch statements both below and in the CheckHeader() function so if new required files are added
be sure to modify code in both places.
*/

switch(FileNo)
{
case 1:
        sprintf(InFile, "%s%s\\treelist_%s.bin",PREFIX,CommonInitial,ENVT);
        break;
case 2:
        sprintf(InFile, "%s%s\\elev_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 3:
        sprintf(InFile, "%s%s\\aspect_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 4:
        sprintf(InFile, "%s%s\\slope_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 5:
        sprintf(InFile, "%s%s\\owner_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 6:
        sprintf(InFile, "%s%s\\pag_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 7:
        sprintf(InFile, "%s%s\\alloc_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 8:
        sprintf(InFile, "%s%s\\minor_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 9:
        sprintf(InFile, "%s%s\\strbuf_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 10:
        sprintf(InFile, "%s%s\\firehist_%s.bin",PREFIX,ConstantInput,ENVT);
        break;
case 11:
        sprintf(InFile, "%s%s\\veg_%s.bin",PREFIX,CommonInitial,ENVT);
        break;
case 12:
        sprintf(InFile, "%s%s\\stage_%s.bin",PREFIX,CommonInitial,ENVT);
        break;
case 13:
        sprintf(InFile, "%s%s\\rule%d_%s.bin",PREFIX,ConstantInput,PRULE,ENVT);
        break;


default:
        Bailout(68);
}
```

```
//Initialize and Re-Initialize the TempInput array to prepare and hold new data
for(r=0;r<ROWS;r++)
{
          for(c=0;c<COLUMNS;c++)
                    TempInput[r][c] = 0;
}

//Check the header data associated with this binary file - program will bail if header file is bad
ND = CheckHeader(FileNo);

//******read in every element of the current binary data and store in the TempInput array
BIN = fopen(InFile, "rb");
if( fread(TempInput,sizeof(TempInput),ROWS*COLUMNS,BIN) != ROWS*COLUMNS)        //TempInput is only a pointer!!
          Bailout(66);
fclose(BIN);


/*
Use the same criteria as reading in the original Cellid.bin file.  If there was a valid value in the TempCellid
spot
then input data from the same spot from TempInput.  Also, check the current value in TempInput and if it has
a NoData value (according to its .hdr file) then put a NODATAFLAG in the spot(because most nodata will be -9999 and
mose of these are ushort which can't handle that value).
*/
cell = 0;                                                  //keep track of which array element to fill
for(r=1;r<=ROWS;r++)
{
          for(c=1;c<=COLUMNS;c++)
          {
                    if(TempCellid[r-1][c-1] != CellidND)              //The original Cellid - if not NoData, then
grab it
                    {
                              //first grab the raw float value
                              RawValue = TempInput[r-1][c-1];

                              //Now switch to the appropriate file so data can be placed properly and conversion
made
                              switch(FileNo)
                              {
                              case 1:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NONFOREST value in
                                                  Data.Treelist[cell] = (ulong)NONFOREST;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Treelist[cell] = (ulong)RawValue;
                                        break;
                              case 2:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NODATAFLAG in
                                                  Data.Elev[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Elev[cell] = (ushort)RawValue;
                                        break;
                              case 3:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NODATAFLAG in
                                                  Data.Aspect[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Aspect[cell] = (ushort)RawValue;
                                        break;
                              case 4:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NODATAFLAG in
                                                  Data.Slope[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Slope[cell] = (ushort)RawValue;
                                        break;
                              case 5:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NODATAFLAG in
                                                  Data.Owner[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Owner[cell] = (ushort)RawValue;
                                        break;
                              case 6:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NODATAFLAG in
                                                  Data.Pag[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Pag[cell] = (ushort)RawValue;
                                        break;
                              case 7:
                                        if(RawValue == ND)                                                  //There is
NoData for this file in this spot, put NODATAFLAG in
                                                  Data.Alloc[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                  Data.Alloc[cell] = (ushort)RawValue;
                                        break;
```

```
                                case 8:
                                        if(RawValue == ND)                                              //There is
NoData for this file in this spot, put NODATAFLAG in
                                                Data.Minor[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                Data.Minor[cell] = (ushort)RawValue;
                                        break;
                                case 9:
                                        if(RawValue == ND)                                              //There is
NoData for this file in this spot, put NODATAFLAG in
                                                Data.Buffer[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                Data.Buffer[cell] = (ushort)RawValue;
                                        break;
                                case 10:
                                        if(RawValue == ND)                                              //There is
NoData for this file in this spot, put NODATAFLAG in
                                                Data.FireHistory[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                Data.FireHistory[cell] = (ushort)RawValue;
                                        break;
                                case 11:
                                        if(RawValue == ND)                                              //There is
NoData for this file in this spot, put NODATAFLAG in
                                                Data.InitialVeg[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                Data.InitialVeg[cell] = (ushort)RawValue;
                                        break;
                                case 12:
                                        if(RawValue == ND)                                              //There is
NoData for this file in this spot, put NODATAFLAG in
                                                Data.InitialStage[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                Data.InitialStage[cell] = (ushort)RawValue;
                                        break;
                                case 13:
                                        if(RawValue == ND)                                              //There is
NoData for this file in this spot, put NODATAFLAG in
                                                Data.PRule[cell] = (ushort)NODATAFLAG;
                                        else
          //RawValue is good, just convert to correct type and put in right spot
                                                Data.PRule[cell] = (ushort)RawValue;
                                        break;


                                default:
                                        Bailout(68);
                                }//end switch

                                //increment array position counter
                                cell++;

                        }//end if(TempCellid[r-1][c-1] != CellidND)
                }
}//end for(r=1;r<=ROWS;r++)

printf("**Binary file %s has been inputted and is OK**\n",InFile);

}//end for(FileNo=0;FileNo<=FILES;FileNo++)


//Delete the TempCellid and TempInput arrays from free store since they is no longer needed
delete [] TempCellid;
delete [] TempInput;




return TRUE;
}//end BinaryReadData


//*****************************************************************************************************
void ReadGoalHoldFound(int Goal)
//*****************************************************************************************************
{
/*
  This function is to read in the binary files for a GOAL-HOLD solution
  from a previous simulation run, and reenter those values into Data.Goal[] & Data.Hold[].
  These binary files were generated in BinarySaveGoalHold, found in "goal_controller.cpp.

  For GROW_ONLY goal, just skip this function because the Goal-Hold values default to 9-0
  during initialization in CreateMainData().
*/

FILE *BIN;
char GoalInFile[256];
char HoldInFile[256];
```

```
ushort *ptr_goal;
ushort *ptr_hold;

//-------------------------------- End of variable defining ---------------------------------------

printf("**Reading binary GOAL and HOLD values from previous simulation.\n");


if(Goal != GROW_ONLY)
{

          //Make the correct output file names
          sprintf(GoalInFile, "%s%s%d\\%s_%s_goal.bin",PREFIX,RerunDir,GOAL_TO_USE,OPTPREFIX,ENVT);
          sprintf(HoldInFile, "%s%s%d\\%s_%s_hold.bin",PREFIX,RerunDir,GOAL_TO_USE,OPTPREFIX,ENVT);


          //Now read them back in
          ptr_goal = &Data.Goal[0];
          ptr_hold = &Data.Hold[0];

          BIN = fopen(GoalInFile, "rb");
          if( BIN == NULL )
                    printf("THERE IS NO OLD GOAL FILE FOR THIS LANDSCAPE - BAILING!!!!\n");
          fread(ptr_goal,sizeof(Data.Goal[0]),UNIQUE,BIN);
          fclose(BIN);

          BIN = fopen(HoldInFile, "rb");
          if( BIN == NULL )
                    printf("THERE IS NO OLD HOLD FILE FOR THIS LANDSCAPE - BAILING!!!!\n");
          fread(ptr_hold,sizeof(Data.Hold[0]),UNIQUE,BIN);
          fclose(BIN);
}


}//end ReadGoalHoldFound

// ****************************************************************************************
long CheckHeader(int File)
// ****************************************************************************************
{
/*
Input code values ("File") and what data they are referring to:

0         Cellid
1         Treelist
2         Elevation
3         Aspect
4         Slope
5         Ownership
6         Plant Association Group
7         Allocation (for federal lands)
8         Minor (sub-watersheds)
9         Buffer (stream buffers on federal lands)
10        Fire History
11        InitialVeg
12        InitialStage
13  Prescription Rule allocation for Framework only
*/

FILE *IN;
char HeaderFile[150];
char garbage[13];
char ByteOrder[10];
int Row,Column;
long Nodata;
double xll, yll, junk;

//Get the appropriate file to check
switch(File)
{
case 0:
          sprintf(HeaderFile, "%s%s\\cellid_%s.hdr",PREFIX,ConstantInput,ENVT);          break;
case 1:
          sprintf(HeaderFile, "%s%s\\treelist_%s.hdr",PREFIX,CommonInitial,ENVT);                  break;
case 2:
          sprintf(HeaderFile, "%s%s\\elev_%s.hdr",PREFIX,ConstantInput,ENVT);                      break;
case 3:
          sprintf(HeaderFile, "%s%s\\aspect_%s.hdr",PREFIX,ConstantInput,ENVT);          break;
case 4:
          sprintf(HeaderFile, "%s%s\\slope_%s.hdr",PREFIX,ConstantInput,ENVT);           break;
case 5:
          sprintf(HeaderFile, "%s%s\\owner_%s.hdr",PREFIX,ConstantInput,ENVT);           break;
case 6:
          sprintf(HeaderFile, "%s%s\\pag_%s.hdr",PREFIX,ConstantInput,ENVT);                       break;
case 7:
          sprintf(HeaderFile, "%s%s\\alloc_%s.hdr",PREFIX,ConstantInput,ENVT);           break;
case 8:
          sprintf(HeaderFile, "%s%s\\minor_%s.hdr",PREFIX,ConstantInput,ENVT);           break;
case 9:
          sprintf(HeaderFile, "%s%s\\strbuf_%s.hdr",PREFIX,ConstantInput,ENVT);          break;
case 10:
          sprintf(HeaderFile, "%s%s\\firehist_%s.hdr",PREFIX,ConstantInput,ENVT);                  break;
case 11:
          sprintf(HeaderFile, "%s%s\\veg_%s.hdr",PREFIX,CommonInitial,ENVT);                       break;
case 12:
```

```
          sprintf(HeaderFile, "%s%s\\stage_%s.hdr",PREFIX,CommonInitial,ENVT);          break;
case 13:
          sprintf(HeaderFile, "%s%s\\rule%d_%s.hdr",PREFIX,ConstantInput,PRULE,ENVT);   break;

default:
          Bailout(68);
}


IN = fopen(HeaderFile, "r");

if(IN == NULL)
          printf("Problem opening HeaderFile %s\n",HeaderFile);

//Now scan in the header data and verify that it is OK
fscanf(IN,"%s %d %s %d %s %lf %s %lf %s %lf %s %ld %s %s",
                                                  garbage, &Column, garbage, &Row, garbage, &xll, garbage, &yll,
                                                  garbage, &junk, garbage, &Nodata, garbage, ByteOrder);


//Check the number of Rows and Columns for this data
if(Column != COLUMNS || Row != ROWS)
          Bailout(69);

//Check the grid origin for this data
if( (int)(xll) != XLL || (int)(yll) != YLL)
          Bailout(70);

//Check to make sure the ByteOrder is correct
if(strcmp(strlwr(ByteOrder),"lsbfirst"))                    //convert to lowercase and test against "lsbfirst" -
which it should be!
          Bailout(71);

fclose(IN);

//write out a little OK line
printf("Header file %s is OK\n",HeaderFile);

//When finished, return the current Nodata value so it can be used during input into the Data.* arrays
return Nodata;
}//end CheckHeader


// ***************************************************************************************
// ***************************************************************************************
//                              Function used regardless of whether BINARY or ASCII method used
// ***************************************************************************************
// ***************************************************************************************
// ***********************************************************
int MakeLink(void)
// ***********************************************************
{
// ================ Create and fill the LINK array for later use ============
/*
This function can be used whether the data was read in ASCII or BINARY format because
it uses the values found in Data.Gridrow, which is populated by either method
*/

int r,c,Start,HowMany;
int *ptr_link;
ushort *ptr_gridrow;

int ctr;
ulong NumberCellid, *ptr_cellid;
//------------------------------ End of variable defining


//Initialize the LINK array - with row numbers 1-N and 0's -- see below for more
ptr_link = &link[0][0];
for(r=1;r<ROWS+1;r++)
{
          *ptr_link = r;
          ptr_link++;
          for(c=0;c<2;c++)
          {
                    *ptr_link = 0;
                    ptr_link++;
          }
}


//Fill the array called Link, which has 3 columns:  The first
//represents the original ROW number and the second has its starting row position
//in any of the Data.* arrays (not the array subscript)and the third column is how
//many values are going into the Data.* arrays for that row.
//The link array will have spaces for ALL
//rows, regardless if data actually gets put in (will have 0's if nodata for the whole row).

Start = 0;
for(r=1;r<=ROWS;r++)
{
          ptr_gridrow = &Data.GridRow[Start];
          ptr_link = &link[r-1][1];
```

```
          if(*ptr_gridrow == (ushort)r)
          {
                    *ptr_link = (Start+1);
                    HowMany = 1;
                    do
                    {
                              ptr_gridrow++;
                              HowMany++;
                    } while ( *ptr_gridrow == (ushort)r );

                    Start = Start + (HowMany-1);
                    *(ptr_link+1) = (HowMany-1);
          }
}


//count the number of values in the CELLID column
for(ctr=0;ctr<UNIQUE;ctr++)
{
          ptr_cellid=&Data.Cellid[ctr];
          if(*ptr_cellid != 0)
                    NumberCellid = ctr+1;
}

printf("There are %lu unique cellids\n",NumberCellid);
printf("First CELLID is: %lu and the last CELLID is: %lu\n",Data.Cellid[0],
                                                                        Data.Cellid[NumberCellid-1]);

//Call up the PrintToStat() function to fill up with the total cell count info.
PrintToStat(1, (ulong)NumberCellid);

return TRUE;
}//end of MakeLink()


// ****************************************************************************
// ****************************************************************************
//                                   Start of functions to handle importing the data in ASCII format
// ****************************************************************************
// ****************************************************************************

// ************************************************************
int AsciiReadCell(void)
// ************************************************************
{

FILE *READ_CELL;
char garbage[13];
int Row,Column;
int r,c;
long int TestValue,Nodata;
ulong ConvertTest,cell;
double xll, yll, junk;
char Temp[150];

//Open the cellid.asc file and check validity
sprintf(Temp, "%s%s\\cellid_%s.asc",PREFIX,ConstantInput,ENVT);

          READ_CELL = fopen(Temp,"r");

          if (READ_CELL == NULL)
                    fprintf(stderr, "opening of %s failed:  %s\n",  Temp, strerror(errno));
          else
#ifdef DEBUG_OPEN1
                    printf("File: %s opened with no problems in mode READ!\n",Temp);
#endif



          fscanf(READ_CELL,"%s %d %s %d %s %lf %s %lf %s %lf %s %ld",
                                                    garbage, &Column, garbage, &Row, garbage, &xll,
garbage, &yll,
                                                    garbage, &junk, garbage, &Nodata);


                    //Do some error checking and bail if input data is not correct
                    if(Column == COLUMNS && Row == ROWS)
                              printf("Rows and columns for CELLID.asc are OK\n");
                    else
                              Bailout(2);


                    //Do some error checking and bail if input data is not correct
                    if( floor(xll) == XLL && floor(yll) == YLL)
                              printf("X and Y origin for CELLID.asc are OK\n");
                    else
                              Bailout(3);
// ============= Enter CELLID data into the maindata array ============================


#ifdef DEBUG_STRUCT
                    //print out the above to see if initialized correctly
                    printf("Cellid\tRow\tColumn\tTlist\tUnit\n");
                    for(r=0;r<UNIQUE;r++)
                    {
```

```
                    printf("%lu\t%hu\t%hu\t%hu\t%hu\n",Data.Cellid[r],Data.GridRow[r],
                                                Data.GridColumn[r],Data.Treelist[r][0],Data.Unit[r]);
          }
#endif

          //Fill the Data.Cellid with valid CELLID values and Data.GridRow with the
          //original ROW cell number and the Data.GridColumn with original COLUMN cell.
          cell = 0;
          for(r=1;r<=ROWS;r++)
          {
                    for(c=1;c<=COLUMNS;c++)
                    {
                              fscanf(READ_CELL,"%ld", &TestValue);
                              if(TestValue != Nodata)
                              {
                                        ConvertTest = (ulong)TestValue;
                                        Data.Cellid[cell] = ConvertTest;        //The CELLID value
                                        Data.GridRow[cell] = (ushort)r;                  //original grid
row
                                        Data.GridColumn[cell] = (ushort)c;      //original column row
                                        cell++;
                              }
                    }
          }

#ifdef DEBUG_STRUCT
          //print out the above to see if filled correctly
          printf("Cellid\tRow\tColumn\tTlist\tUnit\n");
          for(r=0;r<UNIQUE;r++)
          {
                    printf("%lu\t%hu\t%hu\t%hu\t%hu\n",Data.Cellid[r],Data.GridRow[r],
                                                Data.GridColumn[r],Data.Treelist[r][0],Data.Unit[r]);
          }
#endif



fclose(READ_CELL);

return TRUE;
}//end of AsciiReadCell



FUELSTUFF.CPP

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"                    //to hold global DEFINES, etc..
#include "data.h"



//------------ For FUEL stuff
void InitialFuelController(void);
ulong CountLandscape(void);
ulong FillAllInfo(struct P_INFO AllInfo[] );
int CompareAllInfoTreelist(const void *ptr1, const void *ptr2);
ulong CountUniqueAllInfoTree(struct P_INFO AllInfo[], ulong Count);
ulong FillFuelsWithTreelist(struct P_INFO AllInfo[], ulong Count, struct INITIAL_FUELS Fuels[]);
Void InitialFuelLoad(struct INITIAL_FUELS Fuels[], ulong Count);
void InitializeFuelLoadings(struct INITIAL_FUELS Fuels[], int Ctr);
Void DetermineInitialFuelModel(struct INITIAL_FUELS Fuels[], ulong Count);
void LoadInitialFuelModelsAndLoads(struct INITIAL_FUELS Fuels[], ulong Count);
void AdjustFuelStuffForGrowth(int ActualPer);
void FuelDecayAndContribution(int ActualPer);
void CalculateAndFillSimFuelModel(int ActualPer);
void DoubleCheckFuels(void);

void CountTreelistRecords(int HowMany[], char Filename[]);
Void FillTreelistRecords(char Filename[],  struct TREELIST_RECORD Snags[], struct TREELIST_RECORD Live[],
                                                struct TREELIST_RECORD Dwd[], int HowMany[]);
int CompareFuelsForTreelist(const void *ptr1, const void *ptr2);




//Definined in StandData.cpp
extern void CalculateIndividualBasalCanopyWidth(struct TREELIST_RECORD Records[], int NoRecords);
extern void CalculateStandClassification(struct TREELIST_RECORD Records[], int NoRecords, struct STAND_CLASS
*Stand);


//-----------------------------------------------------------------------------------------------
//*************************************************************************************************************
**********
void AdjustFuelStuffForGrowth(int ActualPer)
//*************************************************************************************************************
**********
{
/*
```

```
This may prove to be a time-hog but for now this function will operate on a cell-by-cell basis.  The reason is
that, in
theory, I feel it may be very difficult to track all the potential different fuel loadings by "groups" of cells
that
have identical loadings and other parameters (such as elevation, canopy closure, Qmd, etc.) that are all used to
differentiate into fuel models.
*/
//For Time information
clock_t Start, Finish;
double Duration;

// ------------------------------End of variable defining ----------------------------------------------------

Start = clock();

          printf("Going to calculate fuel decay and contribution !!\n");

          //First, Decay and add on the Net Fuel Loading contributions to each cell
          FuelDecayAndContribution(ActualPer);

          //Now go through and recalculate fuel models for every cell
          CalculateAndFillSimFuelModel(ActualPer);


Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\n\nIt took %.21f seconds to run FuelDecayAndContribution() & CalculateAndFillSimFuelModel()\n", Duration
);
printf("=========================================================================================================
====\n");

}//end AdjustFuelStuffForGrowth


//*************************************************************************************************************
**********
void CalculateAndFillSimFuelModel(int ActualPer)
//*************************************************************************************************************
**********
{
/*
At this point, the fuel loadings should have been adjusted for the current period by FuelDecayAndContribution().
It is
now simply a matter of going to each cell and determining the fuel model using the rules Jim & Bernie developed for
"Reclassificaiton of Fuel Model after the First Period" as seen in Jim's Sept. 17, 1999 documentation.
*/

int a;
int ArrayPer;
int TempCode, TempCover, TempDiam, TempVeg;
int Hour1Fuels, Hour10Fuels, Hour100Fuels;
//-------------------------------- End of variable defining ----------------------------------------

//Set the ArrayPer variable
ArrayPer = ActualPer - 1;

for(a=0;a<UNIQUE;a++)
{

          if(Data.Cellid[a] == FALSE )                                                    //No more to check
                    break;

          if(Data.Treelist[a] == NONFOREST)
                    continue;
          //Go on to next cell - already double checked in FuelDecayAndContribution()

          //Break apart Data.Vegcode to identify the pieces because that is a piece the fuel model matrix needs
          //REMEMBER: The TempCode could have 5 or 10 for TempVeg
          TempCode  = Data.Vegcode[a][ArrayPer];                    //The actual 3 or 4 digit code from PREMO
          TempCover = TempCode%10;                                                              //last digit
for determining stage ( 0 is <= 60%,  1 is > 60% )
          TempDiam  = ( (TempCode-TempCover)%100 ) / 10;    //next to last digit also for determining stage (is the
QMD group)
          TempVeg             = (TempCode-TempCode%100) / 100;                  //1st or 1st two digits for
determining VegCode


          //Now go through the classification matrix
          if( TempDiam == 0 || ( TempDiam <=1 && TempCover == 0) )
          {
                    if( Data.Elev[a] > (3000*FT2M) )
                              Data.FuelModel[a][ArrayPer] = 5;
                    else
                    {
                              if(TempVeg == VC_PINE)
                                        Data.FuelModel[a][ArrayPer] = 2;
                              else if(TempVeg == VC_DH)
                                        Data.FuelModel[a][ArrayPer] = 17;
                              else
                                        Data.FuelModel[a][ArrayPer] = 6;
                    }
          }
          else if( TempVeg == VC_DH && (TempCover == 0 || ( TempDiam <= 1 && TempCover == 1) ) )
                    Data.FuelModel[a][ArrayPer] = 6;
          else
```

```
        {

                //Create the 1,10,100 hour variables to reduce computation and ease of seeing what's going on
                Hour1Fuels                  = Data.Litter[a][ArrayPer] + Data.Class25[a][ArrayPer];
                Hour10Fuels                 = Data.Class1[a][ArrayPer];
                Hour100Fuels        = Data.Class3[a][ArrayPer];


                //REMEMBER:  The Data."fuel load"[][] values are modified ushort - must expand Jims variables
by FUEL_LOAD_EXP to match
                //Also:  Jim's values are in TONS
                if( Hour1Fuels <= 1.5 * FUEL_LOAD_EXP )
                {
                        if( Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                Data.FuelModel[a][ArrayPer] = 18;
                        else if( Hour10Fuels < 4.5 * FUEL_LOAD_EXP )
                                Data.FuelModel[a][ArrayPer] = 8;
                        else
                                Data.FuelModel[a][ArrayPer] = 11;
                }
                else if( Hour1Fuels <= 2.5 * FUEL_LOAD_EXP )
                {
                        if( Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                 Data.FuelModel[a][ArrayPer] = 20;
                        else if( Hour10Fuels < 2 * FUEL_LOAD_EXP )
                        {
                                if( Hour100Fuels <= 1 * FUEL_LOAD_EXP )
                                        Data.FuelModel[a][ArrayPer] = 2;
                                else
                                        Data.FuelModel[a][ArrayPer] = 23;
                        }
                        else if( Hour10Fuels <= 6 * FUEL_LOAD_EXP )
                                Data.FuelModel[a][ArrayPer] = 31;
                        else
                                Data.FuelModel[a][ArrayPer] = 32;
                }
                else
                {
                        if( Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                Data.FuelModel[a][ArrayPer] = 9;
                        else if( Hour10Fuels < 3 * FUEL_LOAD_EXP )
                        {
                                if( Hour100Fuels <= 3.5 * FUEL_LOAD_EXP )
                                        Data.FuelModel[a][ArrayPer] = 16;
                                else
                                        Data.FuelModel[a][ArrayPer] = 10;
                        }
                        else
                                Data.FuelModel[a][ArrayPer] = 12;
                }

        }//end of main else for the class. matrix

}//end for(a=0 ... )

}//end CalculateAndFillSimFuelModel

//*************************************************************************************************************************
**********
void FuelDecayAndContribution(int ActualPer)
//*************************************************************************************************************************
**********
{
/*
This function will take care of decay existing fuel loads and then adding on the Net Contribution of fuel loads to
get
a new fuel load, for each cell.

Because all Fuel Loading values are stored as modified ushort (to maintain some precision) all Data.* values need
to be first
divided by FUEL_LOAD_EXP and then multiplied by TONS before Decaying!  The AfterDecay* (AD*) values can then be
reversed
(divide by TONS and multiply by FUEL_LOAD_EXP) and directly added to the values in Data."fuel load
class"[][ArrayPer]. I will
use a new variable called FACTOR that will do the equivalent.

The decay rates are a combination of those found in  Table 4.6 of
"Fire and Fuels Extension: Model Description - Working Draft", by ESSA Technologies Ltd (Beukema et al.) - Feb 16,
1999.
Also, Jim Agee updated the DUFF and LITTER decay rates in his writeup (dated Sept 17,1999) in Section 6. Additions
to Fire and
Fuels Extension (FFE) of FVS.


*/
int a;
int ArrayPer, ArrayPrevPer;
int AD_Duff, AD_Litter, AD_Class25, AD_Class1, AD_Class3, AD_Class6, AD_Class12, AD_ClassOver12;    //AD is "After
Decay"
int FACTOR, MoveLitter;
//-------------------------- End of variable defining -------------------------------------------------
//printf("Here in FuelDecayAndContribution for period %d\n",ActualPer);

//Set the FACTOR variable so it's not always recalculated
FACTOR = TONS / FUEL_LOAD_EXP;
```

```
//Set the variables for ease of figuring out what array spots to use
ArrayPer = ActualPer - 1;
ArrayPrevPer = ArrayPer - 1;


//Go one-by-one through the Data.* array
for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )                                   //No more to check
                    break;

          if(Data.Treelist[a] == NONFOREST)
          {
                    //Do a quick check and make sure Fuel Models are OK
                    if( Data.FuelModel[a][ArrayPer] == 99 || Data.FuelModel[a][ArrayPer] == 98 ||
                              Data.FuelModel[a][ArrayPer] == 4  || Data.FuelModel[a][ArrayPer] == 19 ||
                              Data.FuelModel[a][ArrayPer] == 1)
                    {}
                    else
                              Bailout(102);

                    continue;                                   //Go on to next cell
          }

          /*************************************************************************************************************
*********
                                                                                                                    Decay
Fuels

                    The AD_* variables will be "reset" after each cell and they represent the current ACTUAL value
of the fuel load.
                    NOTE: All these variables get factored out to LBS-ACRE to help not lose precision.
                    NOTE: is not clear if existing duff should be decayed first then litter stuff added on.  The
way
                    Jim's notes specified to move litter to duff first, then decay but wording was a bit confusing.
          *************************************************************************************************************
*********/
          if( ActualPer == 1)
          {
/*
printf("InitialFuelLoadings (in LBS-ACRE):\n");
printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",Data.InitialDuff[a]*FACTOR,Data.InitialLitter[a]*FACTOR,
             Data.InitialClass25[a]*FACTOR, Data.InitialClass1[a]*FACTOR, Data.InitialClass3[a]*FACTOR,
Data.InitialClass6[a]*FACTOR,
             Data.InitialClass12[a]*FACTOR, Data.InitialClassOver12[a]*FACTOR);
*/

                    //In period 1, the actual fuel loads to decay are really those found in the Initial* arrays
                    AD_Class25                    = (int)( (Data.InitialClass25[a]*FACTOR)           * pow(.88,
YIP) );
                    AD_Class1           = (int)( (Data.InitialClass1[a]*FACTOR)            * pow(.88, YIP) );
                    AD_Class3           = (int)( (Data.InitialClass3[a]*FACTOR)            * pow(.91, YIP) );
                    AD_Class6           = (int)( (Data.InitialClass6[a]*FACTOR)            * pow(.985, YIP) );
                    AD_Class12                    = (int)( (Data.InitialClass12[a]*FACTOR)           * pow(.985,
YIP) );
                    AD_ClassOver12      = (int)( (Data.InitialClassOver12[a]*FACTOR) * pow(.985, YIP) );

                    //The duff & litter are handled a bit differently per Jim Agee.  His rules:
                    //        1) Take 2% of incoming litter and vaporize it - this really means the "existing"
litter - the 2% is for all 5 Years
                    AD_Litter = (int)( (Data.InitialLitter[a]*FACTOR) * .98 );
                    //        2) Then take 1/6 of total litter and move it to duff
                    MoveLitter          = (int)(AD_Litter * .16666);
                    AD_Duff   = MoveLitter;
                    AD_Litter = AD_Litter - MoveLitter;
                    //        3) Decay duff at 3% a year              - REMEMBER to "plus-add" this on
                    AD_Duff   += (int)( (Data.InitialDuff[a]*FACTOR)  * pow(.97,YIP) );

//printf("AfterDecay values for period 1\n");
//printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",AD_Duff, AD_Litter, AD_Class25, AD_Class1, AD_Class3, AD_Class6,
AD_Class12, AD_ClassOver12);
          }
          else
          {
                    //For all other periods, the actual loads to decay are those from the previous period
                    AD_Class25                    = (int)( (Data.Class25[a][ArrayPrevPer]*FACTOR)             *
pow(.88, YIP) );
                    AD_Class1           = (int)( (Data.Class1[a][ArrayPrevPer]*FACTOR)             * pow(.88, YIP)
);
                    AD_Class3           = (int)( (Data.Class3[a][ArrayPrevPer]*FACTOR)             * pow(.91, YIP)
);
                    AD_Class6           = (int)( (Data.Class6[a][ArrayPrevPer]*FACTOR)             * pow(.985,
YIP) );
                    AD_Class12                    = (int)( (Data.Class12[a][ArrayPrevPer]*FACTOR)            *
pow(.985, YIP) );
                    AD_ClassOver12      = (int)( (Data.ClassOver12[a][ArrayPrevPer]*FACTOR)        * pow(.985,
YIP) );

                    //The duff & litter are handled a bit differently per Jim Agee.  His rules:
                    //        1) Take 2% of incoming litter and vaporize it - this really means the "existing"
litter - the 2% is for all 5 Years
                    AD_Litter = (int)( (Data.Litter[a][ArrayPrevPer]*FACTOR) * .98 );
                    //        2) Then take 1/6 of total litter and move it to duff
                    MoveLitter          = (int)(AD_Litter * .16666);
```

```
                    AD_Duff   = MoveLitter;
                    AD_Litter = AD_Litter - MoveLitter;
                    //        3) Decay duff at 3% a year              - REMEMBER to "plus-add" this on
                    AD_Duff   += (int)( (Data.Duff[a][ArrayPrevPer]*FACTOR)      * pow(.97,YIP) );

          }//end else if(ActualPer == 1)
/*
printf("And those fuel loadings to add on are:\n");
printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",Data.Duff[a][ArrayPer]*FACTOR,Data.Litter[a][ArrayPer]*FACTOR,
            Data.Class25[a][ArrayPer]*FACTOR, Data.Class1[a][ArrayPer]*FACTOR, Data.Class3[a][ArrayPer]*FACTOR,
Data.Class6[a][ArrayPer]*FACTOR,
            Data.Class12[a][ArrayPer]*FACTOR, Data.ClassOver12[a][ArrayPer]*FACTOR);
*/
          //*****************************************************************************************************
*******
          //                                                                                    Add Net Contributions
          //*****************************************************************************************************
*******
          //The AD_* variables now have the current After Decay fuel load values in LBS-ACRE.  These should be
converted
          //back to the modified ushort TONS-ACRE and then added on the current modified ushort TONS-ACRE values
that
          //are stored in the current periods net contribution Data."fuel Load"[][] as they were calculated in
Premo;
          //REMEMBER: there are no NEW net contributions to Duff from Premo - it was all handled in above decay
from litter
          Data.Duff[a][ArrayPer]                              = (ushort)(AD_Duff / FACTOR);
          Data.Litter[a][ArrayPer]                = (ushort)Data.Litter[a][ArrayPer]                + (AD_Litter /
FACTOR);
          Data.Class25[a][ArrayPer]               = (ushort)Data.Class25[a][ArrayPer]               + (AD_Class25 /
FACTOR);
          Data.Class1[a][ArrayPer]                = (ushort)Data.Class1[a][ArrayPer]                + (AD_Class1 /
FACTOR);
          Data.Class3[a][ArrayPer]                = (ushort)Data.Class3[a][ArrayPer]                + (AD_Class3 /
FACTOR);
          Data.Class6[a][ArrayPer]                = (ushort)Data.Class6[a][ArrayPer]                + (AD_Class6 /
FACTOR);
          Data.Class12[a][ArrayPer]               = (ushort)Data.Class12[a][ArrayPer]               + (AD_Class12 /
FACTOR);
          Data.ClassOver12[a][ArrayPer] = (ushort)Data.ClassOver12[a][ArrayPer] + (AD_ClassOver12 / FACTOR);
/*
printf("Resulting in these FinalFuelLoadings in modified TONS-ACRE (divide by 10 to get real value) , Note some
precision loss:\n");
printf("%hu\t%hu\t%hu\t%hu\t%hu\t%hu\t%hu\t%hu\n\n",Data.Duff[a][ArrayPer],Data.Litter[a][ArrayPer],
            Data.Class25[a][ArrayPer], Data.Class1[a][ArrayPer], Data.Class3[a][ArrayPer],
Data.Class6[a][ArrayPer],
            Data.Class12[a][ArrayPer], Data.ClassOver12[a][ArrayPer]);
*/

}//end for(a=0 ... )

}//end FuelDecayAndContibution

//*****************************************************************************************************************
**********
void InitialFuelController(void)
//*****************************************************************************************************************
**********
{
ulong ForestCells, SecondForestCount;
ulong Unique, SecondUnique;


//---------------------------------- End of variable defining ----------------------------------------------

printf("\n\n=================================\n");
printf("    Initializing Fuel Loadings      \n");
printf("=================================\n");

ForestCells = CountLandscape();
printf("There are %lu forested cells in this landscape (probably more than in a Solution!)\n",ForestCells);

//Send data out to stat.txt file
PrintToStat(2,(ulong)ForestCells);

//Allocate an array of P_INFO structures to hold each "ForestCells" combination of Treelist-Goal-Hold
struct P_INFO(*AllInfo) = new struct P_INFO[ForestCells];
if(AllInfo == NULL )
          printf("Problems allocating memory for AllInfo[] with %lu records\n",ForestCells);
//Initialize
memset(AllInfo, 0, sizeof( struct P_INFO) * ForestCells );

//Fill up AllInfo with the Treelist Goal Hold for all those cells that are forested
SecondForestCount = FillAllInfo(AllInfo);
if( SecondForestCount != ForestCells )
          Bailout(97);

//Sort those records in AllInfo - use mgsort because there may be alot of records
mgsort( (void*)AllInfo,                                                                    //base
                    ForestCells,                                                                    //count of
records
                    sizeof( struct P_INFO),                                                 //size of each record
                    0,  ForestCells-1,                                                      //current division (
always: 0, "count"-1 )
                    CompareAllInfoTreelist   );                                 //compare function
```

```
//NOTE: At Time 0 - the Goal & Hold do not matter for the initialization of the FUEL LOADINGS - only different
treelist !

//Now go through and count the Unique Treelist in AllInfo
Unique = CountUniqueAllInfoTree(AllInfo, ForestCells);
printf("There were actually %lu UNIQUE Treelist for the forested cells\n",Unique);


//Create a structure to hold new Fuel Loading information for those unique Treelist
struct INITIAL_FUELS(*Fuels) = new struct INITIAL_FUELS[Unique];
if(Fuels == NULL )
        printf("Problems allocating memory for Fuels[] with %lu records\n",Unique);
//Initialize
memset(Fuels, 0, sizeof(struct INITIAL_FUELS) * Unique );

//Now fill up the Fuels structures
SecondUnique = FillFuelsWithTreelist(AllInfo, ForestCells, Fuels);
if(SecondUnique != Unique )
        Bailout(97);

//Can delete the AllInfo structures now they are broken down into Unique records
delete [] AllInfo;


//Get the initial (i.e. default) fuel loadings that will be used for the entire landscape
InitialFuelLoad(Fuels, Unique);

//Now with those fuel loadings, get the initial fuel model assignment at Time 0 - REMEMBER: not used, stands need
to grow to Per 1 first!
DetermineInitialFuelModel(Fuels, Unique);

//Finally, go through the Data.*[] arrays and load in the actual initial fuel model
LoadInitialFuelModelsAndLoads(Fuels, Unique);


//Delete stuff on free store
delete [] Fuels;

//Go through and make sure everyone has fuel loads and model
DoubleCheckFuels();

printf("=============================================================================\n");
printf("    Finished with initial fuels and starting to initialize Background ERA values     \n");
printf("=============================================================================\n\n");

}//end InitialFuelController


//*************************************************************************************************************
void LoadInitialFuelModelsAndLoads(struct INITIAL_FUELS Fuels[], ulong Count)
//*************************************************************************************************************
{
/*
This function will fill up the InitialFuelModel[] and the Intial fuel loading arrays (e.g. InitialDuff,
InitialLitter, etc).
with the data currently stored in Fuels.
*/
        ,
int a,b;
struct INITIAL_FUELS Key;
struct INITIAL_FUELS *ptr_record;
//---------------------------------- End of variable defining ----------------------------------------------
------

//Go through all of Data.*[] and load up the fuel model appropriately
for(a=0;a<UNIQUE;a++)
{
        if(Data.Cellid[a] == FALSE )  //no more cells to check
                break;

        //Always reinitialize the key to make sure no junk in it
        memset(&Key, 0, sizeof(struct INITIAL_FUELS) );

        if( Data.Treelist[a] == NONFOREST )
        {
                //Assign a new fuel model based on Jim Agee's paper "Reclassification of Fuel Model after the
First Period"
                if(Data.InitialVeg[a] == GIS_BARREN)                                    //BARREN - both should be
the same!
                {
                        if(Data.InitialStage[a] != GIS_BARREN)
                                Bailout(59);
                        else
                        {
                                Data.InitialFuelModel[a] = 99;

                                //And all of this cell's FuelModel[][] should be 99
                                for(b=0;b<NP;b++)
                                        Data.FuelModel[a][b] = 99;
                        }
                }
                else if(Data.InitialVeg[a] == GIS_WATER)                                //WATER - both should be
the same!
```

```
                    {
                              if(Data.InitialStage[a] != GIS_WATER)
                                        Bailout(59);
                              else
                              {
                                        Data.InitialFuelModel[a] = 98;

                                        //And all of this cell's FuelModel[][] should be 98
                                        for(b=0;b<NP;b++)
                                                  Data.FuelModel[a][b] = 98;
                              }

                    }
                    else if(Data.InitialVeg[a] == GIS_SHRUB)                              //SHRUB - both should be
the same!
                    {
                              if(Data.InitialStage[a] != GIS_SHRUB)
                                        Bailout(59);

                              if( Data.Elev[a] < (3000*FT2M) )                    //check elevation
                              {
                                        Data.InitialFuelModel[a] = 4;

                                        //And all of this cell's FuelModel[][] should be 4
                                        for(b=0;b<NP;b++)
                                                  Data.FuelModel[a][b] = 4;
                              }
                              else
                              {
                                        Data.InitialFuelModel[a] = 19;

                                        //And all of this cell's FuelModel[][] should be 19
                                        for(b=0;b<NP;b++)
                                                  Data.FuelModel[a][b] = 19;
                              }
                    }
                    else if(Data.InitialVeg[a] == GIS_GRASS)                              //GRASS/FORBS - both
should be the same!
                    {
                              if(Data.InitialStage[a] != GIS_GRASS)
                                        Bailout(59);
                              else
                              {
                                        Data.InitialFuelModel[a] = 1;

                                        //And all of this cell's FuelModel[][] should be 1
                                        for(b=0;b<NP;b++)
                                                  Data.FuelModel[a][b] = 1;
                              }
                    }
                    else
                              Bailout(60);

          }//end if(Data.Treelist == NONFOREST )
          else                //Use the Treelist value to make a key and look for that key->Treelist in the Fuels
structure and populate with that
          {
                    //Always reinitialize the key to make sure no junk in it
                    memset(&Key, 0, sizeof(struct INITIAL_FUELS) );

                    //Make the key
                    Key.Treelist = Data.Treelist[a];

                    //Now do a bsearch for that key on the Fuels structure
                    //NOTE:  The Fuels structure should already be sorted by treelist because it was created from
another treelist-sorted structure
                    ptr_record = (struct INITIAL_FUELS*)bsearch(
                              &Key,
                              (void *)Fuels,
                              (size_t)Count,
                              sizeof( struct INITIAL_FUELS),
                              CompareFuelsForTreelist );

                    if(ptr_record == NULL )
                              Bailout(101);

                    //=============================================================================================
                    // First enter the fuel loads determined earlier
                    //REGARDLESS - always load in the Initial Fuel loadings at this point for all FOREST cells
                    //NOTE:  the fuel loading data in ptr_record-> is already in correct units
                    //=============================================================================================
                    if( (ptr_record->VegClass == VC_MC) && (Data.Elev[a] > (3000*FT2M)) )    //See if a VC_MC type
stand
                    {
                              Data.InitialDuff[a]                    = (ushort)ptr_record->MC_Duff;
                              Data.InitialLitter[a]                  = (ushort)ptr_record->MC_Litter;
                              Data.InitialClass25[a]                 = (ushort)ptr_record->MC_Class25;
                              Data.InitialClass1[a]                  = (ushort)ptr_record->MC_Class1;
                              Data.InitialClass3[a]                  = (ushort)ptr_record->MC_Class3;
                              Data.InitialClass6[a]                  = (ushort)ptr_record->MC_Class6All;
                              Data.InitialClass12[a]                 = (ushort)ptr_record->MC_Class12;
                              Data.InitialClassOver12[a]     = (ushort)ptr_record->MC_ClassOver12;
                    }
                    else
                    {
```

```
                    Data.InitialDuff[a]                     = (ushort)ptr_record->Duff;
                    Data.InitialLitter[a]                   = (ushort)ptr_record->Litter;
                    Data.InitialClass25[a]                  = (ushort)ptr_record->Class25;
                    Data.InitialClass1[a]                   = (ushort)ptr_record->Class1;
                    Data.InitialClass3[a]                   = (ushort)ptr_record->Class3;
                    Data.InitialClass6[a]                   = (ushort)ptr_record->Class6All;
                    Data.InitialClass12[a]                  = (ushort)ptr_record->Class12;
                    Data.InitialClassOver12[a]    = (ushort)ptr_record->ClassOver12;
          }


          //=================================================
          // Then load the fuel model calculated earlier
          //=================================================
          if( ptr_record->VegClass == VC_MC )
          {
                    //If elev > 3000' then need to check the MC_FuelModel
                    if( Data.Elev[a] > (3000*FT2M) )
                    {
                              //Check for the FUEL_FLAG
                              if( ptr_record->MC_FuelModel != FUEL_FLAG )
                                        Data.InitialFuelModel[a] = ptr_record->MC_FuelModel;
                              else
                                        Data.InitialFuelModel[a] = 5;
                    }
                    else
                    {
                              //Check for the FUEL_FLAG
                              if( ptr_record->MC_FuelModel != FUEL_FLAG )
                                        Data.InitialFuelModel[a] = ptr_record->MC_FuelModel;
                              else
                              {
                                        if( ptr_record->VegClass == VC_PINE )
                                                  Data.InitialFuelModel[a] = 2;
                                        else if( ptr_record->VegClass == VC_DH )
                                                  Data.InitialFuelModel[a] = 17;
                                        else
                                                  Data.InitialFuelModel[a] = 6;
                              }
                    }
          }//end if VegClass == VC_MC
          else
          {
                    //First just check for the FUEL_FLAG
                    if( ptr_record->FuelModel != FUEL_FLAG )
                              Data.InitialFuelModel[a] = ptr_record->FuelModel;
                    else
                    {
                              if( Data.Elev[a] > (3000*FT2M) )
                                        Data.InitialFuelModel[a] = 5;
                              else
                              {
                                        if( ptr_record->VegClass == VC_PINE )
                                                  Data.InitialFuelModel[a] = 2;
                                        else if( ptr_record->VegClass == VC_DH )
                                                  Data.InitialFuelModel[a] = 17;
                                        else
                                                  Data.InitialFuelModel[a] = 6;
                              }
                    }
          }//end VegClass != VC_MC


     }//end ELSE treelist = NONFOREST

}//end for(a=0 ... )

}//end LoadInitialFuelModelsAndLoads


//*********************************************************************************************
int CompareFuelsForTreelist(const void *ptr1, const void *ptr2)
//*********************************************************************************************
{

     //Just to typecast them since we aren't actually passing in pointers
     struct INITIAL_FUELS *elem1;
     struct INITIAL_FUELS *elem2;

     elem1 = (struct INITIAL_FUELS *)ptr1;
     elem2 = (struct INITIAL_FUELS *)ptr2;


  if( elem1->Treelist < elem2->Treelist )
       //Compare by Treelist
     return -1;
  if( elem1->Treelist > elem2->Treelist )
     return 1;
  else
                              //FINISHED
               return 0;
}//end CompareFuelsForTreelist
```

```
//*****************************************************************************************************************
void DetermineInitialFuelModel(struct INITIAL_FUELS Fuels[], ulong Count)
//*****************************************************************************************************************
{
/*
The initial fuel model assignment is a bit un-necessary anymore.  Originally, Jim and Bernie developed a matrix
using the VegClass and Structural Stage to determine the initial fuel model at Time 0.  They later developed
a new classification of fuel models for after the first period that was based on fuel loadings (stuff in
the struct Fuels[] ).  However, we don't really need a fuel model at Time 0 because that is the start of the
simulation.  What really happens is that we enter Time 1 and Premo spits out growth and yield data for a period
of 5 yrs after the initial period.  That means there has been harvest and growth and this all occurs PRIOR to
initiating a fire in Time 1.  So fuel models for after Time 0 will get calculated later.  Regardless, this function
is something of a check so we can see what the initial fuel assignment is given the current landscape.

This will use the same rules Jim & Bernie developed for "Reclassificaiton of Fuel Model after the First Period" as
seen in Jim's Sept. 17, 1999 documentation.  This is different than original plan to use the GIS matrix mentioned
earlier.

*/

int a;

//------------------------------------ End of variable defining ----------------------------------------------
------

//The Fuels structure should already be sorted by treelist because it was created from another treelist-sorted
structure
//for(a=0;a<Count;a++)
//        printf("%lu\n",Fuels[a].Treelist);

//The Fuels structure should have the necessary information now to actually determine the Fuel Model per Jim Agee's
rules
for(a=0;a<(signed)Count;a++)
{
        //Jim has a weird rule at first, and the only way I can figure out how to handle it without going through
the entire
        //Data.*[] array and recalculating the the below IF statements is to set a "flag" for this condition.  So
when
        //it comes time to fill up Data.*[] if I get the FUEL_FLAG I know to make another calculation based on
elevation
        if( Fuels[a].Qmd == 0 || (Fuels[a].Qmd <= 1 && Fuels[a].Closure < 60) )
        {
                Fuels[a].FuelModel = FUEL_FLAG;

                if( Fuels[a].VegClass == VC_MC)
                        Fuels[a].MC_FuelModel = FUEL_FLAG;
        }
        else if(Fuels[a].VegClass == VC_DH && (Fuels[a].Closure < 60 || (Fuels[a].Qmd <=1 && Fuels[a].Closure >=
60) ) )
                Fuels[a].FuelModel = 6;        //Don't need a MC_FuelModel because it had to be VC_DH to get
this
        else
        {
                //This should take care of EVERYTHING else that doesn't get coded out above

                //REMEMBER:  The Fuels[].* values are modified ushort - must expand Jims variables by
FUEL_LOAD_EXP to match
                //Also:  Jim's values are in TONS
                //Use the 1,10, & 100 hour fuel loadings
                if( Fuels[a].Hour1Fuels <= 1.5 * FUEL_LOAD_EXP )
                {
                        if( Fuels[a].Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                Fuels[a].FuelModel = 18;
                        else if( Fuels[a].Hour10Fuels < 4.5 * FUEL_LOAD_EXP )
                                Fuels[a].FuelModel = 8;
                        else
                                Fuels[a].FuelModel = 11;
                }
                else if( Fuels[a].Hour1Fuels <= 2.5 * FUEL_LOAD_EXP )
                {
                        if( Fuels[a].Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                Fuels[a].FuelModel = 20;
                        else if( Fuels[a].Hour10Fuels < 2 * FUEL_LOAD_EXP )
                        {
                                if( Fuels[a].Hour100Fuels <= 1 * FUEL_LOAD_EXP )
                                        Fuels[a].FuelModel = 2;
                                else
                                        Fuels[a].FuelModel = 23;
                        }
                        else if( Fuels[a].Hour10Fuels <= 6 * FUEL_LOAD_EXP )
                                Fuels[a].FuelModel = 31;
                        else
                                Fuels[a].FuelModel = 32;
                }
                else
                {
                        if( Fuels[a].Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                Fuels[a].FuelModel = 9;
                        else if( Fuels[a].Hour10Fuels <= 3 * FUEL_LOAD_EXP )
                        {
                                if( Fuels[a].Hour100Fuels <= 3.5 * FUEL_LOAD_EXP )
                                        Fuels[a].FuelModel = 16;
```

```
                                        else
                                                Fuels[a].FuelModel = 10;
                                }
                                else
                                        Fuels[a].FuelModel = 12;
                        }


                        //Now check and see if this is a VC_MC and if so also populate the MC_* stuff
                        if( Fuels[a].VegClass == VC_MC )
                        {
                                //Use the 1,10, & 100 hour fuel loadings
                                if( Fuels[a].MC_Hour1Fuels <= 1.5 * FUEL_LOAD_EXP )
                                {
                                        if( Fuels[a].MC_Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                                Fuels[a].MC_FuelModel = 18;
                                        else if( Fuels[a].MC_Hour10Fuels < 4.5 * FUEL_LOAD_EXP )
                                                Fuels[a].MC_FuelModel = 8;
                                        else
                                                Fuels[a].MC_FuelModel = 11;
                                }
                                else if( Fuels[a].MC_Hour1Fuels <= 2.5 * FUEL_LOAD_EXP )
                                {
                                        if( Fuels[a].MC_Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                                Fuels[a].MC_FuelModel = 20;
                                        else if( Fuels[a].MC_Hour10Fuels < 2 * FUEL_LOAD_EXP )
                                        {
                                                if( Fuels[a].MC_Hour100Fuels <= 1 * FUEL_LOAD_EXP )
                                                        Fuels[a].MC_FuelModel = 2;
                                                else
                                                        Fuels[a].MC_FuelModel = 23;
                                        }
                                        else if( Fuels[a].MC_Hour10Fuels <= 6 * FUEL_LOAD_EXP )
                                                Fuels[a].MC_FuelModel = 31;
                                        else
                                                Fuels[a].MC_FuelModel = 32;
                                }
                                else
                                {
                                        if( Fuels[a].MC_Hour10Fuels < 1 * FUEL_LOAD_EXP )
                                                Fuels[a].MC_FuelModel = 9;
                                        else if( Fuels[a].MC_Hour10Fuels < 3 * FUEL_LOAD_EXP )
                                        {
                                                if( Fuels[a].MC_Hour100Fuels <= 3.5 * FUEL_LOAD_EXP )
                                                        Fuels[a].MC_FuelModel = 16;
                                                else
                                                        Fuels[a].MC_FuelModel = 10;
                                        }
                                        else
                                                Fuels[a].MC_FuelModel = 12;
                                }
                        }//end if( Fuels[a].VegClass == VC_MC )

                }//end of the main ELSE statement

//printf("Just got FuelModel %hu and MC_FuelModel %hu\n",Fuels[a].FuelModel, Fuels[a].MC_FuelModel);

}//end for(a=0 ... )

}//end DetermineFuelModel


//*************************************************************************************************************
void InitialFuelLoad(struct INITIAL_FUELS Fuels[], ulong Count)
//*************************************************************************************************************
{
int a, b, Found;
FILE *Open;
char Temp[250], ActualFile[250];
int HowMany[3];
ulong Treelist, TestTree;
int SnagCount, LiveCount, DwdCount;
struct STAND_CLASS StandClass;
struct STAND_CLASS *ptr_stand;
//----------------------------- End of variable defining -----------------------


//for(a=0;a<(signed)Count;a++)
//        printf("Fuels[%d]:\t%lu\t%hu\t%hu\n",a,Fuels[a].Treelist, Fuels[a].Goal, Fuels[a].Hold);

//Grab the Treelist for each of the records in the UL structures
for(a=0;a<(signed)Count;a++)
{
        //Initialize the HowMany array
        for(b=0;b<3;b++)
                HowMany[b] = 0;

        //Initialize StandClass and its pointer
        ptr_stand = &StandClass;
        memset(ptr_stand, 0, sizeof(struct STAND_CLASS) );

        //Since this is period 0 use all the ORIGINAL treelist files.  Because there were
        //many more initial treelist created than we use, the treelist # has to be used
        //to index the particular file we want.  This is the same method that Premo uses.
```

```
        Treelist  = Fuels[a].Treelist;

        //Create a string to hold the name of the "InitialTreeindex.txt" filename
        sprintf(Temp, "%s%s\\%s",PREFIX,ConstantInput,IT_INDEX);

        //Open the Treeindex.txt file
        Open = fopen(Temp, "r");
        if(Open == NULL )
                fprintf(stderr, "Opening of %s  failed:  %s\n", Temp, strerror(errno));

        //Scroll through the IntialTreeindex and find the current treelist and its actual file pathname
        Found = FALSE;
        while( fscanf(Open, "%lu %s",&TestTree, ActualFile) != EOF )
        {
                if( TestTree == Treelist)               //Have the match
                {
                        //printf("Found %s\n",ActualFile);
                        Found = TRUE;
                        break;
                }
        }//end while....

        //Test to make sure the file was found
        if( Found == FALSE )
                Bailout(98);

        //Close the open file
        fclose(Open);

        //Count how many SNAGS-LIVE-DWD records there are in the treelist
        CountTreelistRecords(HowMany, ActualFile);

        //Allocate free store memory for each of 3 types of TREELIST_RECORD structures
        struct TREELIST_RECORD (*SnagRecords)   = new struct TREELIST_RECORD[HowMany[0]];
        struct TREELIST_RECORD (*LiveRecords)   = new struct TREELIST_RECORD[HowMany[1]];
        struct TREELIST_RECORD (*DwdRecords)    = new struct TREELIST_RECORD[HowMany[2]];

        if(SnagRecords == NULL)
                printf("Problems allocating memory for SnagRecords[] with %d records\n",HowMany[0]);
        if(LiveRecords == NULL)
                printf("Problems allocating memory for LiveRecords[] with %d records\n",HowMany[1]);
        if(DwdRecords == NULL)
                printf("Problems allocating memory for DwdRecords[] with %d records\n",HowMany[2]);

        //Initialize
        memset( SnagRecords, 0, sizeof(struct TREELIST_RECORD) * HowMany[0]);
        memset( LiveRecords, 0, sizeof(struct TREELIST_RECORD) * HowMany[1]);
        memset( DwdRecords, 0, sizeof(struct TREELIST_RECORD) * HowMany[2]);


        //Send off Records to get filled with the treelist data and verify numbers
        FillTreelistRecords(ActualFile, SnagRecords, LiveRecords, DwdRecords, HowMany);

        //Set some counters for how many records are in each type
        SnagCount = HowMany[0];
        LiveCount = HowMany[1];
        DwdCount  = HowMany[2];

        //Get the BASAL AREA and CANOPY WIDTH for the LiveRecords
        CalculateIndividualBasalCanopyWidth(LiveRecords, LiveCount);

        //Get the three items we use in our Veg-Structural classification
        CalculateStandClassification(LiveRecords, LiveCount, ptr_stand);

        //Put the values now found in StandClass into Fuels
        Fuels[a].Basal                  = ptr_stand->Basal;
        Fuels[a].VegClass   = ptr_stand->VegClass;
        Fuels[a].Qmd                    = ptr_stand->Qmd;
        Fuels[a].CoverClass = ptr_stand->CoverClass;
        Fuels[a].Closure    = ptr_stand->Closure;
        //printf("%s has classification %hu%hu%hu and Basal %.2f\n",ActualFile, Fuels[a].VegClass, Fuels[a].Qmd,
Fuels[a].CoverClass, Fuels[a].Basal);

        //Now send data off to initialize the Fuel Loadings
        InitializeFuelLoadings(Fuels, a);


        //Delete stuff on free store
        delete [] SnagRecords;
        delete [] LiveRecords;
        delete [] DwdRecords;

}//end for(a=0 ... )

}//end InitialFuelLoad


//*************************************************************************************************************
****
void InitializeFuelLoadings(struct INITIAL_FUELS Fuels[], int Ctr)
//*************************************************************************************************************
****
{
/*
As far as I can tell, these are the default values that Jim Agee wants to use instead of Table 4.2 of
```

```
"Fire and Fuels Extension: Model Description - Working Draft", by ESSA Technologies Ltd (Beukema et al.) - Feb 16,
1999.
Jim's version is dated 7-27-99 and he labels as Table 4.2 as well.  This is in his section of the writeup
he did (dated Sept 17,1999) in Section 6. Additions to Fire and Fuels Extension (FFE) of FVS.

NOTE:  There was no documentation on what to do with VC_OPEN .  That classificaition is a strange one that I
believe
Lou Beers and Heidi put in Premo without discussing what it meant.  It WAS NOT part of our original veg
classification
scheme and thus Jim Agee did not know to code that one out.  For now, I will assume that VC_OPEN is a product of
those young stands (probably down in the valleys) that have a mix component so I will use DH values for it.??????
*/

ushort VegClass;

//---------------------------------- End of variable defining -------------------------------------------


//Grab the VegClass associated with current stand.  It was calculated earlier in CalculateStandClassification
VegClass = Fuels[Ctr].VegClass;


//************  Assign default fuel loadings depending upon initial vegclass ***********************
if( VegClass == VC_CH || VegClass == VC_MC || VegClass == VC_PINE )
        //1,5,7
{
        Fuels[Ctr].Duff                         = 5.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Litter               = 2.5 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class25              = 0.2 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class1               = 0.8 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class3               = 1.2 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6All            = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part1          = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part2          = 0;
        Fuels[Ctr].Class12              = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].ClassOver12          = 1.0 * FUEL_LOAD_EXP;

        if(VegClass == VC_MC)                   //Parallel variables for MC > 3000'
        //5
        {
                Fuels[Ctr].MC_Duff                      = 10.0 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Litter                    =  1.4 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Class25                   =  0.9 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Class1                    =  2.1 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Class3                    =  3.8 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Class6All                 =  3.0 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Class6Part1       =  3.0 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_Class6Part2       =  0;
                Fuels[Ctr].MC_Class12                   =  9.5 * FUEL_LOAD_EXP;
                Fuels[Ctr].MC_ClassOver12       =  9.5 * FUEL_LOAD_EXP;
        }
}
else if( VegClass == VC_DH )
                                                //2
{
        Fuels[Ctr].Duff                         = 2.3 * FUEL_LOAD_EXP;
        Fuels[Ctr].Litter               = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class25              = 0.3 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class1               = 2.4 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class3               = 5.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6All            = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part1          = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part2          = 0;
        Fuels[Ctr].Class12              = 2.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].ClassOver12          = 2.0 * FUEL_LOAD_EXP;
}
else if( VegClass == VC_EH )
                                                //3
{
        Fuels[Ctr].Duff                         = 3.7 * FUEL_LOAD_EXP;
        Fuels[Ctr].Litter               = 1.8 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class25              = 0.3 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class1               = 1.6 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class3               = 3.1 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6All            = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part1          = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part2          = 0;
        Fuels[Ctr].Class12              = 1.5 * FUEL_LOAD_EXP;
        Fuels[Ctr].ClassOver12          = 1.5 * FUEL_LOAD_EXP;
}
else if( VegClass == VC_KP )
                                                //4
{
        Fuels[Ctr].Duff                         = 5.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Litter               = 2.6 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class25              = 0.3 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class1               = 0.3 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class3               = 0.4 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6All            = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part1          = 1.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].Class6Part2          = 0;
        Fuels[Ctr].Class12              = 2.0 * FUEL_LOAD_EXP;
        Fuels[Ctr].ClassOver12          = 3.0 * FUEL_LOAD_EXP;
}
```

```
else if( VegClass == VC_RF )
                                                //8
{
          Fuels[Ctr].Duff                               = 30.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Litter                   =  0.7 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class25                  =  0.7 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class1                   =  2.6 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class3                   =  3.6 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6All                =  5.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6Part1              =  5.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6Part2              =  0;
          Fuels[Ctr].Class12                  =  4.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].ClassOver12              =  5.0 * FUEL_LOAD_EXP;
}
else if( VegClass == VC_WF )
                                                //9
{
          Fuels[Ctr].Duff                               = 30.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Litter                   =  0.6 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class25                  =  0.8 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class1                   =  2.7 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class3                   =  2.7 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6All                =  4.5 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6Part1              =  4.5 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6Part2              =  0;
          Fuels[Ctr].Class12                  =  5.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].ClassOver12              =  7.0 * FUEL_LOAD_EXP;
}
else if( VegClass == VC_OPEN )                  //John did not have this coded in PREMO - I put same values as
VC_DH so the program wouldn't bail
{
          Fuels[Ctr].Duff                               = 2.3 * FUEL_LOAD_EXP;
          Fuels[Ctr].Litter                   = 1.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class25                  = 0.3 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class1                   = 2.4 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class3                   = 5.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6All                = 1.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6Part1              = 1.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].Class6Part2              = 0;
          Fuels[Ctr].Class12                  = 2.0 * FUEL_LOAD_EXP;
          Fuels[Ctr].ClassOver12              = 2.0 * FUEL_LOAD_EXP;
}
else
          Bailout(100);


//Now group up the needed components to make the 1,10, and 100 hour fuel loadings
Fuels[Ctr].Hour1Fuels                   = Fuels[Ctr].Litter + Fuels[Ctr].Class25;
Fuels[Ctr].Hour10Fuels                  = Fuels[Ctr].Class1;
Fuels[Ctr].Hour100Fuels                 = Fuels[Ctr].Class3;

//Make the parallel groups for when MC
if(VegClass == VC_MC)                   //Parallel variables for MC > 3000'                                     //5
{
          Fuels[Ctr].MC_Hour1Fuels                      = Fuels[Ctr].MC_Litter + Fuels[Ctr].MC_Class25;
          Fuels[Ctr].MC_Hour10Fuels                     = Fuels[Ctr].MC_Class1;
          Fuels[Ctr].MC_Hour100Fuels                    = Fuels[Ctr].MC_Class3;
}

}//End InitializeFuelLoadings

//*****************************************************************************************************************
void FillTreelistRecords(char Filename[],  struct TREELIST_RECORD Snags[], struct TREELIST_RECORD Live[],
                                                        struct TREELIST_RECORD Dwd[], int HowMany[])
//*****************************************************************************************************************
{
FILE *IN;

ushort TestPlot, TestStatus;
int SnagCount, LiveCount, DwdCount;
//---------------------------------- End of variable defining ----------------------

//Open up the filename passed in
IN = fopen(Filename, "r");


SnagCount = 0;
LiveCount = 0;
DwdCount  = 0;
while( fscanf(IN,"%hu %hu ",&TestPlot, &TestStatus) != EOF )
{
          if(TestStatus == SNAG)                        //Put everything in Snags
          {
                    Snags[SnagCount].Plot               = TestPlot;
                    Snags[SnagCount].Status             = TestStatus;

                    //Scan the rest in
                    fscanf(IN, "%f %hu %hu %f %f %f %hu",&Snags[SnagCount].Tpa,&Snags[SnagCount].Model,
&Snags[SnagCount].Report,
                              &Snags[SnagCount].Dbh, &Snags[SnagCount].Height,&Snags[SnagCount].Ratio,
&Snags[SnagCount].Condition);

                    SnagCount++;
          }
          else if(TestStatus == LIVE)           //Put everything in Live
```

```
        {
                Live[LiveCount].Plot                    = TestPlot;
                Live[LiveCount].Status                  = TestStatus;

                //Scan the rest in
                fscanf(IN, "%f %hu %hu %f %f %f",&Live[LiveCount].Tpa, &Live[LiveCount].Model,
&Live[LiveCount].Report,
                        &Live[LiveCount].Dbh, &Live[LiveCount].Height, &Live[LiveCount].Ratio);

                LiveCount++;
        }
        else                                                            //Put everything in Dwd
        {
                Dwd[DwdCount].Plot              = TestPlot;
                Dwd[DwdCount].Status            = TestStatus;

                //Scan the rest in
                fscanf(IN, "%f %hu %hu %f %f %f %hu",&Dwd[DwdCount].Tpa,&Dwd[DwdCount].Model,
&Dwd[DwdCount].Report,
                        &Dwd[DwdCount].Dbh, &Dwd[DwdCount].Height,&Dwd[DwdCount].Ratio,
&Dwd[DwdCount].Condition);

                DwdCount++;
        }
}//end while ...

fclose(IN);

//Error check how many records just went into the different *Records structures
if(SnagCount != HowMany[0])
{
        printf("Number of SnagRecords not matching between ExtractTreelist & FillRecords...bailing\n");
        Bailout(99);
}
if(LiveCount != HowMany[1])
{
        printf("Number of LiveRecords not matching between ExtractTreelist & FillRecords...bailing\n");
        Bailout(99);
}
if(DwdCount != HowMany[2])
{
        printf("Number of Dwdecords not matching between ExtractTreelist & FillRecords...bailing\n");
        Bailout(99);
}

}//end FillRecords


//*****************************************************************************************
void CountTreelistRecords(int HowMany[], char Filename[])
//*****************************************************************************************
{
FILE *Open;
double Plot,Status,Tpa,Model,Report,Dbh,Height,Ratio,Dead;

//---------------------------------- End of variable defining --------------------------------


//Open up the filename passed in
Open = fopen(Filename, "r");

//Start going through and counting records
while( fscanf(Open, "%lf",&Plot) != EOF )         //if Plot is EOF then file end has been reached
{
        //Scan in the next 7 variables
        fscanf(Open,"%lf %lf %lf %lf %lf %lf %lf",&Status,&Tpa,&Model,&Report,&Dbh,&Height,&Ratio);

        //Some Dbh 0's may have slipped in treelist, catch and give them a small dbh
        if(Dbh == 0)
                Dbh = 0.1;

        //For Snags and DWD
        if(Status != LIVE)
                fscanf(Open, "%lf" , &Dead);


        //Tally up the actual records for each "Live" type
        if(Status == SNAG)
                HowMany[0]++;
        else if(Status == LIVE)
                HowMany[1]++;
        else
                HowMany[2]++;

}//end while....


//Close the file
fclose(Open);
}//end CountTreelistRecords

//*****************************************************************************************
ulong CountLandscape(void)
//*****************************************************************************************
{
```

```
/* Count how many FOREST CELLS there are on the landscape - these are all the cells
eligible to receive a prescription.
*/
ulong Count=0;
int a;

//------------------------------ End of variable defining ---------------------------------

for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )
                    break;                                                    //no more data to check

          if(Data.Treelist[a] != NONFOREST )
                    Count++;
}

          return Count;
}//end CountLandscape


//*****************************************************************************************
ulong FillAllInfo(struct P_INFO AllInfo[] )
//*****************************************************************************************
{
//Same thing as CountLandscape() except this time fill up the AllInfo structures
/*
Sometimes this function is called and that is really needed is the Treelist value - but
this function will always fill up Goal & Hold so it is more versatile and if a calling
functions doesn't need them - so be it! (e.g. the initial fuel loading stuff)
*/

int a;
ulong Count=0;

//---------------------------------- End of variable defining ----------------------------

for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )
                    break;                                                    //no more data to check

          if(Data.Treelist[a] != NONFOREST )
          {
                    AllInfo[Count].Treelist                 = Data.Treelist[a];
                    AllInfo[Count].Goal                     = Data.Goal[a];
                    AllInfo[Count].Hold                     = Data.Hold[a];

                    Count++;
          }
}

          return Count;
}//end FillAllInfo

//*********************************************************************************************
int CompareAllInfoTreelist(const void *ptr1, const void *ptr2)
//*********************************************************************************************
{

          //Just to typecast them since we aren't actually passing in pointers
          struct P_INFO *elem1;
          struct P_INFO *elem2;

          elem1 = (struct P_INFO *)ptr1;
          elem2 = (struct P_INFO *)ptr2;


    if( elem1->Treelist < elem2->Treelist )
          //Sort by Treelist
        return -1;
    if( elem1->Treelist > elem2->Treelist )
        return 1;
    else



                    return 0;
                              //FINISHED!!

}//end CompareAllInfo

//*********************************************************************************************
ulong CountUniqueAllInfoTree(struct P_INFO AllInfo[], ulong Count)
//*********************************************************************************************
{
//Go through the array of AllInfo structures and count how many unique TREELIST there are.
ulong a,b;
ulong Unique;
ulong EvalTreelist;
//------------------------ End of variable defining -------------------------------------

Unique = 0;
b = 0;
```

```
for(a=0;a<Count;)                                                    //a will get increment by other
loop
{
          if(b == Count)                                                          //because of weird
incremental method, b will reach end first but a doesn't know that
                    break;

          Unique++;                                                               //first one always counts
as do others because a gets reset in other loop

          //Set the initial Eval* variables
          EvalTreelist        = AllInfo[a].Treelist;


          //since AllInfo is already sorted, start at next record and look downward until no longer a match
          for(b=a+1;b<Count;)
          {
                    if(       AllInfo[b].Treelist == EvalTreelist )
                              b++;
          //look at next record
                    else
                    {
                              //Set the "a" variable to where "b" is because this is the next unique match
                              a = b;
                              break;
                    }
          }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)

          return Unique;
}//end CountUniqueAllInfoTree

//*********************************************************************************************
ulong FillFuelsWithTreelist(struct P_INFO AllInfo[], ulong Count, struct INITIAL_FUELS Fuels[])
//*********************************************************************************************
{

//Same thing as CountUniqueAllInfo, except fill up Fuels at same time

ulong a,b;
ulong Unique;
ulong EvalTreelist;
//------------------------ End of variable defining -------------------------------------

Unique = 0;
b = 0;
for(a=0;a<Count;)                                                    //a will get increment by other
loop
{
          if(b == Count)                                                          //because of weird
incremental method, b will reach end first but a doesn't know that
                    break;

          Unique++;                                                               //first one always counts
as do others because a gets reset in other loop

          //Set the initial Eval* variables
          EvalTreelist        = AllInfo[a].Treelist;

          //Put these in UniqueList
          Fuels[Unique-1].Treelist                = EvalTreelist;


          //since AllInfo is already sorted, start at next record and look downward until no longer a match
          for(b=a+1;b<Count;)
          {
                    if(       AllInfo[b].Treelist == EvalTreelist     )

                              b++;
          //look at next record
                    else
                    {
                              //Set the "a" variable to where "b" is because this is the next unique match
                              a = b;
                              break;
                    }

          }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)

          return Unique;
}//end FillFuelsWithTreelist

//*****************************************************************************************************************
void DoubleCheckFuels(void)
//*****************************************************************************************************************
{
/*
Go through the entire Data.*[] arrays and make sure every cell has some fuel loadings and a valid fuel model
associated with it.
*/
int a;
//------------------------------- End of variable defining ---------------------------------------------

for(a=0;a<UNIQUE;a++)
```

```
{

          if(Data.Cellid[a] == FALSE)
                    break;

          if(Data.Treelist[a] == NONFOREST)
          {
                    //No fuel loads in Nonforest stuff - but check for a fuel model
                    if(Data.InitialFuelModel[a] == 0)
                              printf("Cellid %lu has no fuel model\n",Data.Cellid[a]);
          }
          else      //check both the class3 fuel loading and the fuel model - could also check more FuelLoadings if
wanted
          {
                    if(Data.InitialClass3[a] == 0 || Data.InitialFuelModel[a] == 0 )
                              printf("Cellid %lu has no fuel load and-or fuel model\n",Data.Cellid[a]);
          }
}//end for(a=0 ...)

}//end DoubleCheckFuels


ERASTUFF.CPP

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"                    //to hold global DEFINES, etc..
#include "data.h"


//Functions defined here in EraStuff.cpp
void InitialEraValues(void);
int CalculateSumPeriodEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[], struct
OPTIMIZE_SINGLE_VALUE OV[],


                                                          ulong Records);
void CalculateNetEras( struct CURRENT_ERAS *CellEra );
void CalculateDecayOnlyNetEras( struct CURRENT_ERAS *CellEra );

//External functions
extern int LookAtOSV(const void *ptr1, const void *ptr2);
extern int CompareEraMinor(const void *ptr1, const void *ptr2);
extern int LookAtSolutionCellid(const void *ptr1, const void *ptr2);

//=============================================================================================================
==========

//*************************************************************************************************************
****************
int CalculateSumPeriodEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[], struct
OPTIMIZE_SINGLE_VALUE OV[],


                                                          ulong Records)
//*************************************************************************************************************
****************
{
/*
NOTE:     CS[] is sorted by Cellid in ascending order
                    S_Era[] is sorted by Minor in ascending order
                    OV[] sorted by TREELIST-GOAL-HOLD in ascending order

This functions should get called first after making a random initial solution.  Using that solution fill up and
calculate
every cells ERA value for the simulation period.  For those cells actually in the solution the periodic ERA's are a
function of:

- Their InitialEra
- ERA recovery
- New ERA contribution due to harvesting
- continuing recovery

For those cells NOT actually in the current solution, the periodic ERA values are a function of:

- Their InitialEra
- ERA recovery

To do this:  look at every cell and do a BSEARCH to see if its subwatershed is in solution.  If so, get the
solution
TREELIST-GOAL-HOLD and BSEARCH those values in OV[].  Once found calculate periodic ERA's as noted below and store
in
the S_ERA structure and CS[] structure.


Another STRATEGY note (because I keep forgetting how this works in long run):
The CS[] structure contains a member called CS.PeriodEra[] that will hold the individual contribution that a cell
makes to the
overall Subwatershed ERA value stored in the S_Era.SumPeriodEra[].  Until a final solution is found there is no
need to
store individual ERA values in the permanent Data.Era[] array because it will change during the solution finding
process.  So
```

```
this function is very important because it goes through the entire Data.* arrays and accounts for EVERY cell that
IS in the
solution AND for every cell that is in a subwatershed that IS in the solution.  Those are two distinct things.  The
S_Era.SumPeriodEra[] values recognize that some cells contribute to the subwatershed ERA value even though they are
not
in solution.
In the end, as a solution is being found by the heuristic, when a "move" is made (which only involves moving Goal-
Hold values
from cells that are IN the solution) the contribution that a cell made that is being moved OUT of the solution can
be
subtracted by finding its individual contribution in CS.PeriodEra[] and subtracting that from S_Era.SumPeriodEra[]
and the
new move contribution can be found by taking the new prescription (Treelist-Goal-Hold) and find that prescription
in
the OV[] structure and recalculate the same as is done here -> and added to the S_Era.SumPeriodEra[] and restore in
the
CS.PeriodEra[].
*/

int a,b;
int InSolution;

//Keys and pointers for structures
struct ERA Key;
struct ERA *ptr_record;
struct SOLUTION SKey;
struct SOLUTION *ptr_skey;
struct OPTIMIZE_SINGLE_VALUE OVKey;
struct OPTIMIZE_SINGLE_VALUE *ptr_ovkey;
struct CURRENT_ERAS CellEraValues, *ptr_cev;


//For Time information
clock_t Start, Finish;
double Duration;

//---------------------------------- End of variable defining ----------------------------------------------------
----
printf("Calculating the SumPeriodEra[] and CS[].PeriodEra[] values for this solution\n");


//Always zero out the values in S_Era[].SumPeriodEra[] at start since they are += and may get called multiple times
for(a=0;a<(signed)NoSub;a++)
{
          for(b=0;b<NP;b++)
                    S_Era[a].SumPeriodEra[b] = 0;
}


Start = clock();


for(a=0;a<UNIQUE;a++)
{
          if( Data.Cellid[a] == FALSE )                              //no more cells to check
                    break;

          //NOTE:  This starts off the same way as Fill_SEra() does

          //Since there are no restrictions such as not counting wilderness, every cell has a contribution to
cumulative ERA
          //as long as its subwatershed is in the solution.  Make a key with the subwatershed ID and search for it
          Key.Minor = Data.Minor[a];

          /*
          A cell ALWAYS contributes to the S_Era[].SumPeriodicEra[] values if its "parent" subwatershed is in
S_Era[].Minor.
          The tricky part is to track whether a particular cell is being "managed" (i.e. in the solution) because
its
          SumPeriodicEra[] values are calculated differently. */

          //Use bsearch on S_Era to see if this subwatershed is in solution
          ptr_record = (struct ERA*)bsearch(
                    &Key,
                    (void *)S_Era,
                    (size_t)NoSub,
                    sizeof( struct ERA),
                    CompareEraMinor );

          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++
          //++++++++++++++++++++++++++++++++++ SUBWATERSHED IS NOT IN SOLUTION
++++++++++++++++++++++++++++++++++++++++++++++++++
          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++
          if( ptr_record == NULL )
          {
                    /*
                    There are basically a couple of reasons that this subwatershed is not in the solution:
                    1) It just isn't!  For example, a subwatershed with only wilderness will most likely not be in
the solution
                    2) Something has gone wrong.

                    Do a double check by verifying that the actual Cellid is not in CS[].
                    */
```

```
                    //First, verify that this cellid is not in the solution (should not since it's parent
subwatershed was not!)
                    //There shouldn't be many subwatershed NOT in solution so this should not take up too much
processing time

                    //Make a key for the current cell using its cellid
                    SKey.Cellid                 = Data.Cellid[a];

                    //Use bsearch on CS[] to see if this cell is in the solution
                    ptr_skey  = (struct SOLUTION*)bsearch(
                              &SKey,
                              (void *)CS,
                              (size_t)Count,
                              sizeof( struct SOLUTION),
                              LookAtSolutionCellid );

                    if( ptr_skey != NULL )                          //if it finds this key in CS then something
is wrong!
                              Bailout(103);

          }//end if( ptr_record == NULL )
          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++
          //++++++++++++++++++++++++++++++++++++++++++++ SUBWATERSHED IS IN SOLUTION
++++++++++++++++++++++++++++++++++++++++++++
          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++
          else
          {
                    /*
                    So this cell's parent subwatershed IS in the solution, but the cell itself may not be.  Two
things need to happen:
                    1) Determined whether the cell itself is in the solution and
                    2) account for this cells contribution to the S_Era[].SumPeriodEra[] & CS[].PeriodEra[]
                       since it's parent subwatershed is in solution.
                    */

                    //First, determine whether or not this cellid is in the solution

                    //Make a key for the current cell using its cellid
                    SKey.Cellid                 = Data.Cellid[a];

                    //Use bsearch on CS[] to see if this cell is in the solution
                    ptr_skey  = (struct SOLUTION*)bsearch(
                              &SKey,
                              (void *)CS,
                              (size_t)Count,
                              sizeof( struct SOLUTION),
                              LookAtSolutionCellid );

                    //Make a flag to use below
                    if( ptr_skey == NULL )
                              InSolution = FALSE; //cell not in solution
                    else
                              InSolution = TRUE;

          //*****************************************************************************
                                                                                                        //
AND CELL IS NOT IN SOLUTION

          //*****************************************************************************
                    if( InSolution == FALSE )
                    {
                              /*
                              Since there is no possibility of activity taking place
                              in this cell, just slowly decay or "recover" it's current Data.InitialEra[]
proportionally down to 0.
                              There is no documentation to do this but it should not matter because they don't
contribute to anything.
                              I am thinking that later we may want to "recover" certain areas at different rates
and track how those
                              subwatershed that are "unmanaged" fair compare to those that are managed.

                              This cell will still contribute to the S_Era.SumPeriodEra[] values, but how they are
calculated is different
                              than if it was in solution because this cell (not being in the solution) cannot have
activities and so there are
                              no new net increases in ERA values.
                              */
                              //clear the CellEraValues stuff before filling and sending off
                              memset(&CellEraValues, 0, sizeof(struct CURRENT_ERAS) );

                              //Make a package of stuff to send off to get NetEra's calculated
                              CellEraValues.CurrentEra      = ( (float)Data.InitialEra[a] / ERA_EXP ) ;
          //last stored as modified ushort
                              CellEraValues.Cell                        = a;

                              //Need to send a pointer to get values back
                              ptr_cev = &CellEraValues;

                              //Ship pointer off to function which will calculate DecayOnly NetEra's for each
period
                              CalculateDecayOnlyNetEras(ptr_cev);
```

```
                                //If new decayed NetEras were calculated, store their contribution to the
SumPeriodEra[]
                                if( ptr_cev->NeedsDecay == TRUE )
                                {
                                        for(b=0;b<NP;b++)
                                                ptr_record->SumPeriodEra[b] += (ulong)(ptr_cev->NetEra[b]);
                                }

                        }//end if( InSolution == FALSE )


        //*****************************************************************************
                                                                                        //
AND CELL IS IN SOLUTION

        //*****************************************************************************
                        else
                        {
                                /*
                                Finally, down to the nitty-gritty.  This cell is in the solution and its parent
subwatershed is in solution.
                                The S_Era.SumPeriodEra[] and CS[].PeriodEra[] values are therefore a function of
recovery and net addition due to
                                activities occuring in a period.
                                */
                                //Make a key for this cells Treelist-Goal-Hold as seen in the CS[] structures
                                OVKey.Treelist      = ptr_skey->Treelist;
                                OVKey.Goal                  = ptr_skey->Goal;
                                OVKey.Hold                  = ptr_skey->Hold;


                                //Use bsearch on OV[] to access this prescriptions Rev and CFHarvest values
                                ptr_ovkey = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
                                        &OVKey,
                                        (void *)OV,
                                        (size_t)Records,
                                        sizeof( struct OPTIMIZE_SINGLE_VALUE),
                                        LookAtOSV );

                                if( ptr_ovkey == NULL )                 //there had better be one!
                                {
                                        printf("Can't find key: Treelist = %lu, Goal = %hu, and Hold =
%hu\n",OVKey.Treelist,OVKey.Goal,OVKey.Hold);
                                        Bailout(80);
                                }


                                //clear the CellEraValues stuff before filling and sending off
                                memset(&CellEraValues, 0, sizeof(struct CURRENT_ERAS) );

                                //Make a package of stuff to send off to get NetEra's calculated
                                CellEraValues.ptr_osv                   = ptr_ovkey;
                                CellEraValues.CurrentEra        = ( (float)Data.InitialEra[a] / ERA_EXP ) ;
        //last stored as modified ushort

                                //Need to send a pointer to get values back
                                ptr_cev = &CellEraValues;

                                //Ship pointer off to function which will calculate NetEra's for each period
                                CalculateNetEras(ptr_cev);

                                //Store the return values in the NetEra[] member in two places for each period
                                for(b=0;b<NP;b++)
                                {
                                        ptr_skey->PeriodEra[b]                  = (ushort)(ptr_cev->NetEra[b]);
                                        ptr_record->SumPeriodEra[b] += (ulong)(ptr_cev->NetEra[b]);
                                }

                                //Also store Data.InitialEra[] in the solution structure - is needed when making
moves in heuristic process
                                ptr_skey->InitialEra                    = Data.InitialEra[a];
        //double check how this get used later!!

                        }//end else if( InSolution ... )
                }//end else if( ptr_record == NULL )
}//end for(a=0 ... )


//Testprint
/*
printf("The SumPeriodEra values here in CalculateSumPeriodEra are\n");
for(a=0;a<(signed)NoSub;a++)
{
        printf("Subwatershed %hu has Count %lu:   ",S_Era[a].Minor, S_Era[a].Count);

        for(b=0;b<NP;b++)
                printf("\t%.2f",((float)S_Era[a].SumPeriodEra[b] / ERA_EXP) / S_Era[a].Count);

        printf("\n");

}
*/

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
```

```
printf("!!Took %.2lf seconds to calculate SumPeriodEra[] and CS.PeriodEra[]for the entire landscape**\n", Duration
);

return TRUE;

}//end CalculateSumPeriodEra

//*****************************************************************************
void InitialEraValues(void)
//*****************************************************************************
{
/*
Give background ERA values to all of the landscape - including those that are NONFOREST
For now there are only a few "rules" that give different background levels.  I am
completely making these rules up - based on some values I have seen in the draft document,
"Eldorado National Forest:  Cumulative Off-Site Watershed Effects (CWE) Analysis Process" version 1.1
dated June, 1993.

There is plenty of room here to develop new rules and I will implement those at later dates as more
information is given to me regarding what background values are appropriate
*/

int a;

//---------------------------- End of variable defining -----------------------------------

//Go through all of Data.*[]
for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )  //no more cells to check
                    break;

          if(Data.Minor[a] == WATER_BODY )                              //Data.InitialEra[] already initialized to
0.0 - just skip and leave at 0
                    continue;

          if( Data.Alloc[a] == ALLOC_WILD )                             //Data.InitialEra[] already initialized to
0.0 - just skip and leave at 0
                    continue;

          if( Data.InitialVeg[a] == GIS_WATER )             //Data.InitialEra[] already initialized to 0.0 - just
skip and leave at 0
                    continue;

          //REMEMBER:  I made this up with some "guidance" from document
          if( Data.Buffer[a] == IN_BUFFER )
                              //These should have low background values
                    Data.InitialEra[a] = (ushort)( .01 * ERA_EXP );
          else if( Data.Owner[a] == OWN_PI )
                              //Private Industrial should be highest
                    Data.InitialEra[a] = (ushort)( .08 * ERA_EXP );
          else if( Data.Alloc[a] == ALLOC_MATRIX )
                    //Assume all Matrix lands have had previous activity
                    Data.InitialEra[a] = (ushort)( .08 * ERA_EXP );
          else if( Data.Owner[a] == OWN_STATE || Data.Owner[a] == OWN_MISC )             //State and misc lands
get fairly high value
                    Data.InitialEra[a] = (ushort)( .07 * ERA_EXP );
          else
                    Data.InitialEra[a] = (ushort)( .05 * ERA_EXP );
          //all remaining - give moderate value

}//end for(a=0 ...)

printf("=====================================================\n");
printf("    Finished initializing Background ERA values     \n");
printf("=====================================================\n\n");


}//end InitialEraValues



//*****************************************************************************
void CalculateDecayOnlyNetEras( struct CURRENT_ERAS *CellEra )
//*****************************************************************************
{
/*
This function will calculate the Net Period Era's for any cell that is being DECAYED only.
That is, there is definately NO activity going on in it.  That may be due to it not being
in the solution at all, or if the cells parent subwatershed is in the solution this cell
may still not be in the solution.

NOTE:  The entire structure CellEra was zero'ed out before being called so it is safe to assume
that the NetEra[] array is zero at start.
*/

int b;
float Subtract, LastEra;
//---------------------------- End of variable defining ---------------------------------

//NOTE: could code so this function is not called if a cell has these next attributes, but is easier to do only
once here!
//Determine if one of those cells that received an InitialEra of 0, if so then NetEra[] is fine
if(Data.Minor[CellEra->Cell] == WATER_BODY )                            //NetEra[] already initialized to 0.0 - just
return and leave at 0
```

```
        return;

if( Data.Alloc[CellEra->Cell] == ALLOC_WILD )                          //NetEra[] already initialized to 0.0 - just
return and leave at 0
        return;

if( Data.InitialVeg[CellEra->Cell] == GIS_WATER )          //NetEra[] already initialized to 0.0 - just return and
leave at 0
        return;

//If cell passes the above break statements, set the flag to tell calling function that new non-zero values are in
NetEra
CellEra->NeedsDecay = TRUE;

//Otherwise, "recover" proportionally from its Data.InitialEra[] value
LastEra    = CellEra->CurrentEra;                                      //CurrentEra has the Data.InitialEra[] value
for this cell
Subtract = ( CellEra->CurrentEra / NP * 2);       //This is rounding, but that's OK - the * 2 is because initial
ERA are low!


for(b=0;b<NP;b++)
{
        LastEra = LastEra - Subtract;

        if(LastEra < 0 )
                LastEra = 0;

        CellEra->NetEra[b] = LastEra * ERA_EXP;

}//end for(b=0 ... )

}//end CalculateDecayOnlyNetEras

//*****************************************************************************
void CalculateNetEras( struct CURRENT_ERAS *CellEra )
//*****************************************************************************
{
/*
This function will calculate Net Period Eras for any cell , for all periods.  The Era
coefficients used here are a mixture of stuff. See comment at start of InitialEraValues()
for source of most Era data.
*/

int b;
float CurrentEra;
int LastCutPer, UseAlternate;
float EraSitePrep, EraHarvest, EraRecovery, ThisPeriodHSP, LastPeriodHSP;


//------------------------------ End of variable defining ----------------------------------


//=================================================
//          Calculate Net losses and additions
//=================================================

LastCutPer = -1;

for(b=0;b<NP;b++)
{
        //reset some variables every period
        EraSitePrep                     = 0;
        EraHarvest                      = 0;
        UseAlternate        = FALSE;
        if(b > 0 )
        {
                LastPeriodHSP = ThisPeriodHSP;
                ThisPeriodHSP = 0;
        }
        else      //first period only
        {
                ThisPeriodHSP = 0;
                LastPeriodHSP = 0;
        }


        //******************** RECOVERY *****************************
        //First there is always some "recovery" from previous period
        if( b == 0 )
                EraRecovery         = (float)(CellEra->CurrentEra / NP * 2);          //recover the same as
above stuff
        else
        {
                //Recovery is function of last time there was harvest - these values could easily be modified
if not working
                if( (b - 1) == LastCutPer )
        //last harvest was 1 period ago
                        EraRecovery = (float).08;
                else if( (b - 2) == LastCutPer )
        //last harvest was 2 period ago
                        EraRecovery = (float).02;
                else if( (b - 3) == LastCutPer )
        //last harvest was 3 period ago
                        EraRecovery = (float).02;
```

```
                else
                        EraRecovery         = (float)(CellEra->CurrentEra / NP * 2);
        //recover the same as above stuff

        }


        //******************* NEW ADDITIONS ****************************
        if( CellEra->ptr_osv->CFHarvest[b] > 0 )                    // There was Harvest this period
        {

        //=================================================================================================
                //ERA for SitePrep a function of Revenue generated that period (per discussion with Klaus
Barber)

        //=================================================================================================
                if( CellEra->ptr_osv->Rev[b] < 500 )
                        EraSitePrep = 0;
                else if( CellEra->ptr_osv->Rev[b] > 2500 )
                        EraSitePrep = (float)0.1;
                else                                                //scale the ERA for those revenues between
500 - 2500 dollars from 0.0 - 0.1
                        EraSitePrep = (float)( (CellEra->ptr_osv->Rev[b] - 500) * .00005);// .00005 is .1 /
2000 with 2000 being the range of $ values



        //=================================================================================================
                //ERA for Harvest from pg. 89 of SNEP Addendum (chap 2)... ERA=max(0.01 * mbf, .08) <= 0.2

        //=================================================================================================

                //NOTE:  CFHarvest value needs to be converted to MCF and then using conversion of 5 to get MBF
                EraHarvest = (float)(0.01 * ( .001 * CellEra->ptr_osv->CFHarvest[b] * 5 ));

                if( EraHarvest < 0.08 )
                        EraHarvest = (float)0.08;
                //must be at least 0.08
                else if( EraHarvest > 0.2 )
                        EraHarvest = (float)0.2;
                //but not greater than 0.20

                /*
                Because of problem where a stand may be cut in multiple periods with increasing EraHarvest and
EraSitePrep
                values, the CurrentEra must be controlled or it skyrockets.  To compensate, calculate the total
contribution
                from EraHarvest & EraSitePrep, if it is higher that the previous periods calculations (assuming
it was
                also harvested in previous period) take the difference of the two and add ONLY that difference
on to
                CurrentEra and don't add any EraRecovery.
                */
                ThisPeriodHSP = EraHarvest + EraSitePrep;

                if( LastCutPer == (b-1) )
                        UseAlternate = TRUE;

                LastCutPer = b;

        }//end if( ptr_ovkey-> ... )


        if( UseAlternate == TRUE)
        {
                if( ThisPeriodHSP > LastPeriodHSP )
                        CurrentEra = CurrentEra + (ThisPeriodHSP - LastPeriodHSP);
                else
                        CurrentEra = CurrentEra;
        }
        else //OK, calculate a new CurrentEra by subtracting Recovery values and adding SitePrep and Harvest
values
                CurrentEra = CurrentEra - EraRecovery + EraHarvest + EraSitePrep ;

        //Don't let the CurrentEra "recover" itself below 0 - or constrict it to never be less than its
InitialEra[]
        if(CurrentEra < 0 )
                CurrentEra = 0;

        //Store that value in the NetEra[] member
        CellEra->NetEra[b]  = (CurrentEra * ERA_EXP);

}//end for(b=0 ... )

}//end CalculateNetEras


OUTPUTDATA.CPP

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
```

```
#include "data.h"
#include "goals.h"


//defined here in OutputData.cpp
int OutputCurrentLandscapeData(int Per);
int OutputPreSimAnalysisData(void);
int OutputPostSimAnalysisData(void);
void OutputAcresHarvest(int Status);
void OutputMapGoals(int Status);
void OutputForestDistribution(int Status);
void LevelOfActivity(int Status);
void TimingChoiceFrequency(void);
void OwnershipByMinor(int USW, int UniqueMinor[]);
void OutputFuelLoadsModel(int Per);
void OutputPotentialBigTreesAllStandGoals(void);
void OutputEndingSolutionMetrics(void);
void OutputVegcodes(int Per);
void OutputInitialGoal(void);

//defined in ReadData.cpp
extern int CreateMainData(void);

//defined in FuelStuff.cpp
extern ulong CountLandscape(void);
extern ulong FillAllInfo(struct P_INFO AllInfo[] );
extern int CompareAllInfoTreelist(const void *ptr1, const void *ptr2);
extern ulong CountUniqueAllInfoTree(struct P_INFO AllInfo[], ulong Count);


// ************************************************************
int OutputPreSimAnalysisData(void)
// ************************************************************
{
/* This function doesn't care whether optimization was done at the Watershed or Subwatershed level.
It will output data to use in either making maps in ArcInfo or for making graphs in Excel - as well
as providing general statistical data regarding various elements we want to look at.

There are several pieces of data that can be extracted for a Pre-Simulation analysis.  This controlling
function can be used to 'toggle' which ones we want, because in the end, in may be faster to run
some simulations without doing all these analysis and outputting.
*/

printf("\n\t\t\t\t====== Generating data for PRE-SIMULATION analysis =========\n");

//For Time information
clock_t Start, Finish;
double Duration;

//**********************  Some function Calling Controled by using #defines in global.h *******************
Start = clock();

OutputFuelLoadsModel(0);

OutputForestDistribution(PREDICTED);                        //This should get done no matter what!

//NOTE:  output of initial ERA values is done in GoalController.cpp with PrintInitialEraValues()

if(GOAL_TO_USE != GROW_ONLY)                                //Grow Only - don't need harvest and acres or goal
{

        #ifdef ACRES_HARVEST
                LevelOfActivity(PREDICTED);
                OutputAcresHarvest(PREDICTED);                                          //Ascii file that can be
used in Excel
        #endif

        #ifdef MAP_GOALS
                OutputMapGoals(PREDICTED);                                              //Grid format file for
ArcInfo
        #endif

}//end if(GOAL_TO)USE != GROW_ONLY)


Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\n**It took %.2lf seconds to output the PRE-SIMULATION anylysis data**\n", Duration );


        return TRUE;
}//end OutputPreSimAnalysisData(void);

// ************************************************************
int OutputPostSimAnalysisData(void)
// ************************************************************
{
/*
This function is pretty much identical to the above OutputPreSimAnalysisData(), except that
this will send a parameter 'ACTUAL' to various ouput functions so they know to ouput
results in the ..\PostSimData\goal*\ directories.

This function doesn't care whether optimization was done at the Watershed or Subwatershed level.
It will output data to use in either making maps in ArcInfo or for making graphs in Excel - as well
as providing general statistical data regarding various elements we want to look at.
```

```
There are several pieces of data that can be extracted for a Post-Simulation analysis.  This controlling
function can be used to 'toggle' which ones we want, because in the end, in may be faster to run
some simulations without doing all these analysis and outputting.
*/

printf("\n\t\t\t\t====== Generating data for POST-SIMULATION analysis ========\n");

//For Time information
clock_t Start, Finish;
double Duration;

//**********************  Function Calling Controled by using #defines in global.h ******************
Start = clock();


OutputEndingSolutionMetrics();

OutputForestDistribution(ACTUAL);                              //This should get done no matter what!


if(GOAL_TO_USE != GROW_ONLY)                                   //Grow Only - don't need harvest and acres or goal
{
          #ifdef ACRES_HARVEST
                    LevelOfActivity(ACTUAL);
                    OutputAcresHarvest(ACTUAL);                                   //Ascii file that can be used in
Excel
          #endif

          #ifdef MAP_GOALS                                     //DON'T NEED UNTIL WE COME UP WITH RE-OPTIMIZING
ROUTINES DURING SIMULATION - SAME AS PRE-SIM
                    //OutputMapGoals(ACTUAL);                                             //Grid format file for
ArcInfo
          #endif
}//end if(GOAL_TO)USE !=GROW_ONLY)



Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\n**It took %.2lf seconds to output the POST-SIMULATION anylysis data**\n", Duration );


          return TRUE;
}//end OutputPostSimAnalysisData(void);

// ************************************************************
void OutputEndingSolutionMetrics(void)
// ************************************************************
{
/*
This function will only get called in OutputPostSimAnalysisData() and it is
designed to first create a SOLUTION structure that is identical to the solution used
for this goal - and then fill it up with values as seen below in code.
*/
FILE *WriteOut;
char filename[256];

int a, b;
ulong AllocOK, AllocNOK,CellsInShed;
ulong SolutionCounters[3];

double PerBigTrees[NP];
double SumBigTrees = 0;
ulong *ptr_cellid;
ushort *ptr_bigtrees;
int  FoundMatch;
ulong Cellid;
ulong SolutionSheds;

//------------------------------------ End variable defining ---------------------------------------------
printf("\n\n*** Calculating new Solution Metrics (e.g. EraValues, BigTrees) - will take a few moments ***\n\n");

//*************************************************************************************************************
//                                                                                    Determine the "Solution"
//
// This creates a "bogus" solution but allows this function to use the Solution structure for calls
// to other functions that want that type of structure as a parameter.
//*************************************************************************************************************

//Initialize the SolutionCounters array and call up the DetermineEligibleCells() function to fill it up
for(a=0;a<3;a++)
          SolutionCounters[a] = 0;

if( DetermineEligibleCells(SolutionCounters) == FALSE)
          Bailout(82);

//The values now in SolutionCounters should be properly set
AllocOK             = SolutionCounters[0];
AllocNOK  = SolutionCounters[1];
CellsInShed         = SolutionCounters[2];

//Set a checker to look for when there are 0 eligible cells
if(AllocOK == FALSE)
          Bailout(89);
```

```
//Create an array of structures on the free store to hold the solution
struct SOLUTION (*Solution) = new struct SOLUTION[AllocOK];
if( Solution == NULL )
        printf("Problems allocating memory for Solution[] with %lu elements\n",AllocOK*sizeof(SOLUTION));

//Initialize
memset( Solution, 0, sizeof(struct SOLUTION) * AllocOK );

//Now fill that array of SOLUTION structures with the Treelist - Minor - Cellid - GOAL - and HOLD of those eligible
cells
if( FillSolution(SolutionCounters, Solution, FAKE) == FALSE )
        Bailout(83);

//Now sort the array of SOLUTION structures by MINOR .  This will guarantee all the subwatersheds are in order
//Use mgsort because qsort takes way too long since there are not many unique Minor ID's
mgsort(   (void*)Solution,                                          //base
                (size_t)AllocOK,                                            //count of # of arrays
                sizeof(struct SOLUTION),                            //size of each array
                0, AllocOK-1,                                                       //current division (
always: 0, "Count"-1 )
                LookAtSolutionMinor );                                      //compare function
//=========================================End of defining and filling Solution ======================================

//*************************************************************************************************
//                                                                                                 Determine and print out the ERA
values
//*************************************************************************************************

//Call up the CountSolutionWatersheds() function to see how many subwatersheds are actually in the solution
SolutionSheds = CountSolutionWatersheds(AllocOK, Solution);

if( SolutionSheds == FALSE )
        Bailout(84);

//Create the appropriate number of Solution_ERA structures and store them in an array
struct ERA (*S_Era) = new struct ERA[SolutionSheds];
if(S_Era == NULL)
        printf("Problems allocating memory for S_Era with %lu elements\n",SolutionSheds*sizeof(struct ERA));

//Initialize this array of ERA structures - this is important because Fill_SEra will do some += summing
memset( S_Era, 0, sizeof(struct ERA) * SolutionSheds );


//Fill the array of S_Era structures with appropriate values                    NOTE: the ERA values are ready to
be printed after this!
if( FillEndingEra(SolutionSheds, S_Era, AllocOK, Solution) == FALSE )
        Bailout(85);

//Print out the ERA values in S_Era
PrintSolutionEraValues(S_Era,SolutionSheds, LAST);

//================================== end of doing ERA stuff ============================================


/*      ***************************************************************************************************
                                                                                                        BIG
TREES
        Since the Solution structure was filled during the above call to FillSolution() I can use that to access
        those cells that were actually in the solution.   Remember that the Solution structure gets filled with
        a Treelist, Goal, & Hold but they don't mean anything here.  All that matters is the Cellid and then
        to find that Cellid in the Data.*[] arrays and count up the stored Data.BigTrees[][], which should be
        an accurate reflection of what was on the landscape during each period.

        ***************************************************************************************************
*/
        printf("Counting up the ENDING # of Big Trees for this goal and solution\n\n");


//Resort the  array of Solution structures by CELLID
qsort(    (void*)Solution,                                                  //base
                (size_t)AllocOK,                                                        //count of # of
arrays
                sizeof(struct SOLUTION),                                    //size of each array
                LookAtSolutionCellid        );                                  //compare
function

//Initialize the PerBigTrees[] array
for(a=0;a<NP;a++)
        PerBigTrees[a] = 0;


//Put pointers at start of Data.* arrays
ptr_cellid                  = &Data.Cellid[0];
ptr_bigtrees        = &Data.BigTrees[0][0];

for(a=0;a<(signed)AllocOK;a++)
{
        //REMEMBER- this works because both Solution and Data.Cellid have cellid's in "row/column" order

        //Get Values for current cell in CS
        Cellid              = Solution[a].Cellid;

        //Start looking through the Data.* arrays and find a match
        FoundMatch = 0;
```

```
        do{
                if( *ptr_cellid == Cellid )                                     //Ok, the cellid's match, so should
everything else!
                {
                        for(b=0;b<NP;b++)
                        {
                                PerBigTrees[b] += ((*ptr_bigtrees) * ACREEQ);
                                ptr_bigtrees++;                                                         //on
last period - this will bump the pointer to period 1 of next cell
                        }
                                FoundMatch = 1;
                }

                //increment cellid pointer, whether or not a match was found
                ptr_cellid++;

                //increment ptr_bigtrees only if no match found yet
                if(FoundMatch == 0 )
                        ptr_bigtrees+=NP;

        )while(FoundMatch == 0);

)//end for(a ...)


//Add up the total sum of big trees
for(b=0;b<NP;b++)
        SumBigTrees += PerBigTrees[b]/BIGTREES_EXP;

// ======================== PRINT OUT STUFF BELOW ==================================
//Create and open the file
sprintf(filename, "%s%s%d\\BigTrees.txt",PREFIX,PostSimOutputDir,GOAL_TO_USE);
WriteOut = fopen(filename, "w");

fprintf(WriteOut,"\nThe Periodic Big Trees Totals are:\n");
for(a=0;a<NP;a++)
        fprintf(WriteOut,"Per%d is %-.3lf\n",a+1,PerBigTrees[a]/BIGTREES_EXP);

fprintf(WriteOut,"\n\nThe total sum of Big Trees is:  %.3lf\n",SumBigTrees);
fprintf(WriteOut,"Which amounts to about %.3lf per acre\n",SumBigTrees/(AllocOK*ACREEQ));

fclose(WriteOut);


//Delete stuff on free store
delete [] Solution;
delete [] S_Era;



)//end OutputEndingSolutionMetrics

// *************************************************************
void OutputFuelLoadsModel(int Per)
// *************************************************************
{
/*
This will output a file that has the following columns:

Treelist  Duff      Litter    Class25    Class1     Class3   Class6  Class12   ClassOver12   Pag  Elev  VegCode
FuelModel

Since this is just an analysis function to help Bernie and Jim look at how their fuel model classification scheme
using fuel loads is working, I have two options:  1) print out data for every individual cell in the entire
landscape, or
2) print out data for all unique combinations of the above columns.  Neither of those choices are good.  With over
two million cells for the entire landscape, #1 is just too unwieldy, and #2 would take too much time and I don't
feel like
coding in.  So a compromise:  I will randomly pick a # and use that as my starting point in the Data.*[] arrays and
just
output data for the next 50 cells - then pick another random # and do the same for another 50 cells.  So this will
always
output data for 100 cells across the landscape.  I don't think it's important that we know exactly where they are
but that
could be accounted for if wanted later (using the GridRow and GridColumn[] arrays ).

*/
char Index[250], ActualFile[250], OutFile[250];
FILE *Open, *WriteOut;

ulong a,rnd,Found;
int ArrayPer, Elev;
ulong TestTree;
char *Name;
char SepChar[3]="\\\\";
char LastName[20];
//------------------------------------ End of variable defining ---------------------------------------

printf("\n\n=============================================\n");
printf("    Outputting Fuel Loads and Fuel Model      \n");
printf("=============================================\n\n");

//Set the ArrayPer
ArrayPer = Per - 1;
```

```
//Create a string to hold the name of the "InitialTreeindex.txt" filename
sprintf(Index, "%s%s\\%s",PREFIX,ConstantInput,IT_INDEX);

//Create and open the output file stuff
sprintf(OutFile, "%s%s\\goal%d\\Period_%d_FuelLoads.txt",PREFIX,GeneralDataDir,GOAL_TO_USE,Per);
WriteOut = fopen(OutFile, "w");                                          //No error checking!


//Print out some misc. stuff
fprintf(WriteOut, "The fuel loading values are in TONS-ACRE\n");
fprintf(WriteOut, "There may be some discrepancy with the Treelist type and what was used to initialize loadings in
Period 0\n");
fprintf(WriteOut, "Elev:   0 is < 3000' and 1 is >= 3000'\n");
fprintf(WriteOut, "This is an output for Period %d\n\n\n",Per);

//print out column labels
fprintf(WriteOut, "TreelistFile\tDuff       Litter      Class25      Class1      Class3      Class6      Class12
ClassOver12   Pag    Elev    Vegcode    FuelModel\n");
fprintf(WriteOut,
"=============================================================================================================
==============================\n");



//Get a random # to use for the first 50 cells
do{
        rnd = (ulong)(rand() % UNIQUE);
}while(rnd > UNIQUE - (UNIQUE/2) );                        //Make it way less because of how many Nodata cells
that there are


//Now go through the Data.*[] array for 50 cells starting at cell "rnd"
for(a=rnd;a<rnd+50;a++)
{
        if(Data.Cellid[a] == FALSE)
                break;


        //Get the Elev variable
        if(Data.Elev[a] > (3000*FT2M) )
                Elev = 1;
        else
                Elev = 0;

        //*****************************************************************
        //For period 0 analysis, get the original treelist actually used - not the Treelist #
        if(Per == 0 )
        {

                //Open the Treeindex.txt file
                Open = fopen(Index, "r");
                if(Open == NULL )
                        fprintf(stderr, "Opening of %s  failed:  %s\n", Index, strerror(errno));

                //Scroll through the IntialTreeindex and find the current treelist and its actual file pathname
                Found = FALSE;
                while( fscanf(Open, "%lu %s",&TestTree, ActualFile) != EOF )
                {
                        if( TestTree == Data.Treelist[a])                //Have the match
                        {
                                Found = TRUE;
                                break;
                        }
                }//end while....

                //Test to make sure the file was found
                if( Found == FALSE )
                        Bailout(98);

                //Close the file
                fclose(Open);


                //Extract off the last piece of ActualFile to tell what the treelist is actually for
                Name = strtok(ActualFile,SepChar);
                while(Name != NULL )
                {
                        sprintf(LastName, "%s",Name);
                        Name = strtok(NULL,SepChar);
                }


                fprintf(WriteOut, "%s\t",LastName);
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialDuff[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialLitter[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass25[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass1[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass3[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass6[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass12[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClassOver12[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%hu\t",Data.Pag[a]);
                fprintf(WriteOut, "%d\t",Elev);
                fprintf(WriteOut, "0\t");
                                //There was no Vegcode stored for the initial data!
```

```
                fprintf(WriteOut, "%hu\n",Data.InitialFuelModel[a]);
        }
        else
        {
                if(Data.Treelist[a] < FIRST_AVAILABLE_TREELIST)
                {
                        //Open the Treeindex.txt file
                        Open = fopen(Index, "r");
                        if(Open == NULL )
                                fprintf(stderr, "Opening of %s  failed:  %s\n", Index, strerror(errno));

                        //Scroll through the IntialTreeindex and find the current treelist and its actual
file pathname
                        Found = FALSE;
                        while( fscanf(Open, "%lu %s",&TestTree, ActualFile) != EOF )
                        {
                                if( TestTree == Data.Treelist[a])               //Have the match
                                {
                                        Found = TRUE;
                                        break;
                                }
                        }//end while....

                        //Test to make sure the file was found
                        if( Found == FALSE )
                                Bailout(98);

                        //Close the file
                        fclose(Open);


                        //Extract off the last piece of ActualFile to tell what the treelist is actually for
                        Name = strtok(ActualFile,SepChar);
                        while(Name != NULL )
                        {
                                sprintf(LastName, "%s",Name);
                                Name = strtok(NULL,SepChar);
                        }


                        fprintf(WriteOut, "%s\t",LastName);
                }
                else
                        fprintf(WriteOut, "%hu\t\t ",Data.Treelist[a]);



                //Do all of this regardless of treelist #
                fprintf(WriteOut, "%-12.1f",((float)Data.Duff[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Litter[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Class25[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Class1[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Class3[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Class6[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Class12[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.ClassOver12[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%hu\t",Data.Paq[a]);
                fprintf(WriteOut, "%d\t",Elev);
                fprintf(WriteOut, "%hu\t",Data.Vegcode[a][ArrayPer]);

                fprintf(WriteOut, "%hu\n",Data.FuelModel[a][ArrayPer]);
        }

}//end for(a=rnd ... )

//============================================== The second random fifty
================================================

//Get a random # to use for the second 50 cells
do{
        rnd = (ulong)(rand() % UNIQUE);
}while(rnd > UNIQUE - (UNIQUE/2) );                     //Make it way less because of Nodata cells that there
are


//Now go through the Data.*[] array for 50 cells starting at cell "rnd"
for(a=rnd;a<rnd+50;a++)
{
        if(Data.Cellid[a] == FALSE)
                break;


        //Get the Elev variable
        if(Data.Elev[a] > (3000*FT2M) )
                Elev = 1;
        else
                Elev = 0;

        //*****************************************************************
        //For period 0 analysis, get the original treelist actually used - not the Treelist #
        if(Per == 0 )
        {

                //Open the Treeindex.txt file
                Open = fopen(Index, "r");
```

```
                if(Open == NULL )
                        fprintf(stderr, "Opening of %s  failed:  %s\n", Index, strerror(errno));

                //Scroll through the IntialTreeindex and find the current treelist and its actual file pathname
                Found = FALSE;
                while( fscanf(Open, "%lu %s",&TestTree, ActualFile) != EOF )
                {
                        if( TestTree == Data.Treelist[a])               //Have the match
                        {
                                Found = TRUE;
                                break;
                        }
                }//end while....

                //Test to make sure the file was found
                if( Found == FALSE )
                        Bailout(98);

                //Close the file
                fclose(Open);


                //Extract off the last piece of ActualFile to tell what the treelist is actually for
                Name = strtok(ActualFile,SepChar);
                while(Name != NULL )
                {
                        sprintf(LastName, "%s",Name);
                        Name = strtok(NULL,SepChar);
                }


                fprintf(WriteOut, "%s\t",LastName);
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialDuff[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialLitter[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass25[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass1[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass3[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass6[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClass12[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.InitialClassOver12[a] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%hu\t",Data.Pag[a]);
                fprintf(WriteOut, "%d\t",Elev);
                fprintf(WriteOut, "0\t");
                                //There was no Vegcode stored for the initial data!
                fprintf(WriteOut, "%hu\n",Data.InitialFuelModel[a]);
        }
        else
        {
                if(Data.Treelist[a] < FIRST_AVAILABLE_TREELIST)
                {
                        //Open the Treeindex.txt file
                        Open = fopen(Index, "r");
                        if(Open == NULL )
                                fprintf(stderr, "Opening of %s  failed:  %s\n", Index, strerror(errno));

                        //Scroll through the IntialTreeindex and find the current treelist and its actual
file pathname
                        Found = FALSE;
                        while( fscanf(Open, "%lu %s",&TestTree, ActualFile) != EOF )
                        {
                                if( TestTree == Data.Treelist[a])               //Have the match
                                {
                                        Found = TRUE;
                                        break;
                                }
                        }//end while....

                        //Test to make sure the file was found
                        if( Found == FALSE )
                                Bailout(98);

                        //Close the file
                        fclose(Open);


                        //Extract off the last piece of ActualFile to tell what the treelist is actually for
                        Name = strtok(ActualFile,SepChar);
                        while(Name != NULL )
                        {
                                sprintf(LastName, "%s",Name);
                                Name = strtok(NULL,SepChar);
                        }


                        fprintf(WriteOut, "%s\t",LastName);
                }
                else
                        fprintf(WriteOut, "%hu\t\t ",Data.Treelist[a]);


                //Do all of this regardless of treelist #
                fprintf(WriteOut, "%-12.1f",((float)Data.Duff[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Litter[a][ArrayPer] / FUEL_LOAD_EXP));
                fprintf(WriteOut, "%-12.1f",((float)Data.Class25[a][ArrayPer] / FUEL_LOAD_EXP));
```

```
                    fprintf(WriteOut, "%-12.1f",((float)Data.Class1[a][ArrayPer] / FUEL_LOAD_EXP));
                    fprintf(WriteOut, "%-12.1f",((float)Data.Class3[a][ArrayPer] / FUEL_LOAD_EXP));
                    fprintf(WriteOut, "%-12.1f",((float)Data.Class6[a][ArrayPer] / FUEL_LOAD_EXP));
                    fprintf(WriteOut, "%-12.1f",((float)Data.Class12[a][ArrayPer] / FUEL_LOAD_EXP));
                    fprintf(WriteOut, "%-12.1f",((float)Data.ClassOver12[a][ArrayPer] / FUEL_LOAD_EXP));
                    fprintf(WriteOut, "%hu\t",Data.Pag[a]);
                    fprintf(WriteOut, "%d\t",Elev);
                    fprintf(WriteOut, "%hu\t",Data.Vegcode[a][ArrayPer]);

                    fprintf(WriteOut, "%hu\n",Data.FuelModel[a][ArrayPer]);
          }

}//end of the second for(a=rnd ... )


fclose(WriteOut);

}//end OutputFuelLoadsModel

// ************************************************************
int OutputCurrentLandscapeData(int Per)
// ************************************************************
{
//NOTE:  the incoming "Per" is the correct period to which this data goes (not array subscript)

//NOTE 18NOV99:  I would eventually like to get Finney to rewrite Farsite and Flammap to input binary files
//He has expressed that he could do it later but for now all must be Ascii files


printf("\n====== Outputting landscape data (Fuel, BLC, CBD, Stand Height, and Closure) for period %d...PRE-FIRE!
=========\n",Per);


//Variables for writing the output files
FILE *WRITE_BLC, *WRITE_CBD, *WRITE_HEIGHT, *WRITE_FUEL, *WRITE_CLOSURE;
char BLCFile[256], CBDFile[256], HeightFile[256], FuelFile[256], ClosureFile[256];


int *ptr_srp;                 //Starting Row Position
ushort   *ptr_column;
int r,c,HowMany;
int ColumnsLeft, ctr;
ushort StartColumn,OutColumn;
ushort OutClosure;
ushort *ptr_blc, *ptr_cbd, *ptr_height, *ptr_fuel, *ptr_closure;

//For Time information
clock_t Start, Finish;
double Duration;
//=======================================end variables ============================================================

Start = clock();

//Make the correct output file names
sprintf(BLCFile,      "%s%s%d\\per%d\\blc.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(CBDFile,      "%s%s%d\\per%d\\cbd.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(HeightFile,   "%s%s%d\\per%d\\height.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(FuelFile,     "%s%s%d\\per%d\\fuel.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);
sprintf(ClosureFile,  "%s%s%d\\per%d\\closure.asc",PREFIX,INPUTS,GOAL_TO_USE,Per);

//open up the files to write to
WRITE_BLC =                   fopen(BLCFile, "w");
WRITE_CBD =                   fopen(CBDFile, "w");
WRITE_HEIGHT =      fopen(HeightFile, "w");
WRITE_FUEL =        fopen(FuelFile, "w");
WRITE_CLOSURE = fopen(ClosureFile, "w");

if (WRITE_BLC == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", BLCFile, strerror(errno));
if (WRITE_CBD == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", CBDFile, strerror(errno));
if (WRITE_HEIGHT == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", HeightFile, strerror(errno));
if (WRITE_FUEL == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", FuelFile, strerror(errno));
if (WRITE_CLOSURE == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", ClosureFile, strerror(errno));



//write out the header data to each of the files
fprintf(WRITE_BLC,"ncols\t\t%d\n",COLUMNS);
fprintf(WRITE_BLC,"nrows\t\t%d\n",ROWS);
fprintf(WRITE_BLC,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WRITE_BLC,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WRITE_BLC,"cellsize\t%d\n",CELLSIZE);
fprintf(WRITE_BLC,"NODATA_value\t%d\n",NODATA);

fprintf(WRITE_CBD,"ncols\t\t%d\n",COLUMNS);
fprintf(WRITE_CBD,"nrows\t\t%d\n",ROWS);
fprintf(WRITE_CBD,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WRITE_CBD,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WRITE_CBD,"cellsize\t%d\n",CELLSIZE);
fprintf(WRITE_CBD,"NODATA_value\t%d\n",NODATA);
```

```
fprintf(WRITE_HEIGHT,"ncols\t\t%d\n",COLUMNS);
fprintf(WRITE_HEIGHT,"nrows\t\t%d\n",ROWS);
fprintf(WRITE_HEIGHT,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WRITE_HEIGHT,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WRITE_HEIGHT,"cellsize\t%d\n",CELLSIZE);
fprintf(WRITE_HEIGHT,"NODATA_value\t%d\n",NODATA);

fprintf(WRITE_FUEL,"ncols\t\t%d\n",COLUMNS);
fprintf(WRITE_FUEL,"nrows\t\t%d\n",ROWS);
fprintf(WRITE_FUEL,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WRITE_FUEL,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WRITE_FUEL,"cellsize\t%d\n",CELLSIZE);
fprintf(WRITE_FUEL,"NODATA_value\t%d\n",NODATA);

fprintf(WRITE_CLOSURE,"ncols\t\t%d\n",COLUMNS);
fprintf(WRITE_CLOSURE,"nrows\t\t%d\n",ROWS);
fprintf(WRITE_CLOSURE,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WRITE_CLOSURE,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WRITE_CLOSURE,"cellsize\t%d\n",CELLSIZE);
fprintf(WRITE_CLOSURE,"NODATA_value\t%d\n",NODATA);


for(r=1;r<=ROWS;r++)
{
        ptr_srp = &link[r-1][1];
        HowMany = *(ptr_srp+1);
        StartColumn =                   Data.GridColumn[(*ptr_srp)-1];                                          //not
a pointer!
        ptr_column =                    &Data.GridColumn[(*ptr_srp)-1];
        ptr_blc =                       &Data.HLC[(*ptr_srp)-1][Per-1];
        ptr_cbd =                       &Data.CBDensity[(*ptr_srp)-1][Per-1];
        ptr_height =                    &Data.StandHeight[(*ptr_srp)-1][Per-1];
        ptr_fuel =                              &Data.FuelModel[(*ptr_srp)-1][Per-1];
        ptr_closure                      =      &Data.Closure[(*ptr_srp)-1][Per-1];

        //If the whole row is blank, print out NODATA and goto next row
        if( *ptr_srp == FALSE )                 //means a zero was left in this spot during MakeLink
        {
                for(c=1;c<=COLUMNS;c++)
                {
                        fprintf(WRITE_BLC,"%d  ",NODATA);
                        fprintf(WRITE_CBD,"%d  ",NODATA);
                        fprintf(WRITE_HEIGHT,"%d  ",NODATA);
                        fprintf(WRITE_FUEL,"%d  ",NODATA);
                        fprintf(WRITE_CLOSURE,"%d  ",NODATA);

                }
                //put in new lines
                fprintf(WRITE_BLC, "\n");
                fprintf(WRITE_CBD, "\n");
                fprintf(WRITE_HEIGHT, "\n");
                fprintf(WRITE_FUEL, "\n");
                fprintf(WRITE_CLOSURE, "\n");

                continue;           //goto next row
        }


        //print out NODATA for those cells before data starts
        for(c=1;c<StartColumn;c++)
        {
                fprintf(WRITE_BLC,"%d  ",NODATA);
                fprintf(WRITE_CBD,"%d  ",NODATA);
                fprintf(WRITE_HEIGHT,"%d  ",NODATA);
                fprintf(WRITE_FUEL,"%d  ",NODATA);
                fprintf(WRITE_CLOSURE,"%d  ",NODATA);
        }

        //set some counters
        OutColumn = StartColumn;
        ctr = 0;

        //print out values for area on landscape by checking
        //value in Data.GridColumn to match it with OutColumn value
        do{
                if(*ptr_column == OutColumn)
                {
                        fprintf(WRITE_BLC, "%hu  ",*ptr_blc);
                        fprintf(WRITE_CBD, "%.2f  ",(float)*ptr_cbd / DENSITY_EXP);
                        fprintf(WRITE_HEIGHT, "%hu  ",*ptr_height);
                        fprintf(WRITE_FUEL, "%hu  ",*ptr_fuel);


                        //Check Data.Closure[][] and reclassify data into the 4 categories that Farsite wants
before writing out
                        if(*ptr_closure <= 10)
                                OutClosure = 1;
                        else if(*ptr_closure > 10 && *ptr_closure < 50)
                                OutClosure = 2;
                        else if(*ptr_closure >= 50 && *ptr_closure < 80)
                                OutClosure = 3;
                        else
                                OutClosure = 4;
```

```
                        fprintf(WRITE_CLOSURE, "%hu  ",OutClosure);


                        ptr_blc+=NP;
                        ptr_cbd+=NP;
                        ptr_height+=NP;
                        ptr_fuel+=NP;
                        ptr_closure+=NP;

                        ptr_column++;
                        OutColumn++;
                        ctr++;
                }
                else    //print out NODATA for the "gaps"
                {
                        fprintf(WRITE_BLC,"%d  ",NODATA);
                        fprintf(WRITE_CBD,"%d  ",NODATA);
                        fprintf(WRITE_HEIGHT,"%d  ",NODATA);
                        fprintf(WRITE_FUEL,"%d  ",NODATA);
                        fprintf(WRITE_CLOSURE,"%d  ",NODATA);

                        OutColumn++;
                }
        }while(ctr != HowMany );

        //Check to see how many columns are left to do
        ColumnsLeft = COLUMNS - (OutColumn-1);

        if(ColumnsLeft == 0)
        {
                fprintf(WRITE_BLC, "\n");
                fprintf(WRITE_CBD, "\n");
                fprintf(WRITE_HEIGHT, "\n");
                fprintf(WRITE_FUEL, "\n");
                fprintf(WRITE_CLOSURE, "\n");

                continue;                       //go to next row
        }

        //print out NODATA for those cells after the data that are left
        for(c=0;c<ColumnsLeft;c++)
        {
                fprintf(WRITE_BLC,"%d  ",NODATA);
                fprintf(WRITE_CBD,"%d  ",NODATA);
                fprintf(WRITE_HEIGHT,"%d  ",NODATA);
                fprintf(WRITE_FUEL,"%d  ",NODATA);
                fprintf(WRITE_CLOSURE,"%d  ",NODATA);

        }

        //put in a new line
        fprintf(WRITE_BLC, "\n");
        fprintf(WRITE_CBD, "\n");
        fprintf(WRITE_HEIGHT, "\n");
        fprintf(WRITE_FUEL, "\n");
        fprintf(WRITE_CLOSURE, "\n");

}//end of for(r=1;r<=ROWS;r++)


fclose(WRITE_BLC);
fclose(WRITE_CBD);
fclose(WRITE_HEIGHT);
fclose(WRITE_FUEL);
fclose(WRITE_CLOSURE);

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\n**It took %.2lf seconds to output this periods BLC, CBD, HEIGHT, FUEL and CLOSURE files**\n", Duration );

return TRUE;

}//end OutputCurrentLandscapeData


// ************************************************************
void OutputAcresHarvest(int Status)
// ************************************************************
{

//Variable for writing the output files
FILE *WriteExcel;
char ExcelFile[150];


//Acre counters (number of cells)
ulong NonForestCells = 0, TotalCellCount = 0;
ulong GoalCells[GOALS], CellsTouched[NP], FedTouched[NP], NonFedTouched[NP];
double FedHarvest[NP], NonFedHarvest[NP],TotalFedHarvest=0, TotalNonFedHarvest=0;


int r,c;
ulong *ptr_cellid, *ptr_treelist;
ushort *ptr_goal, *ptr_owner, *ptr_minor, *ptr_buffer;
float *ptr_harvest;
```

```
//=====================================end variables ============================================================

//Make some output filenames and open files
if(Status == PREDICTED)
          sprintf(ExcelFile, "%s%s%d\\acres_harvest.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
else
          sprintf(ExcelFile, "%s%s%d\\acres_harvest.txt",PREFIX,PostSimOutputDir,GOAL_TO_USE);

WriteExcel =        fopen(ExcelFile, "w");
if (WriteExcel == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", ExcelFile, strerror(errno));


//Initialize arrays
for(r=0;r<GOALS;r++)
          GoalCells[r] = 0;

for(r=0;r<NP;r++)
{
          CellsTouched[r] = 0;
          FedTouched[r] = 0;
          NonFedTouched[r] = 0;
          FedHarvest[r] = 0;
          NonFedHarvest[r] = 0;
}



//Start at beginning of Data.*[] arrays and keep tally of items to output.
for(r=0;r<UNIQUE;r++)
{
          //set pointers
          ptr_cellid =        &Data.Cellid[r];
          ptr_treelist =      &Data.Treelist[r];
          ptr_goal =                  &Data.Goal[r];
          ptr_owner =                 &Data.Owner[r];
          ptr_minor =                 &Data.Minor[r];
          ptr_buffer =        &Data.Buffer[r];
          ptr_harvest =       &Data.CFHarvest[r][0];                              //values stored are a PER ACRE
value

          if( *ptr_cellid == FALSE)                                   //no more records to check in array
                    break;

          TotalCellCount++;

          //set an error checker
          if( *ptr_harvest > 0 && *ptr_treelist == NONFOREST)
                    Bailout(53);

          //First, lets track how many acres were assigned to each goal
          if( *ptr_treelist == NONFOREST)                                                   //These were NON-FOREST,
so track separately
                    NonForestCells++;
          else
                    GoalCells[*ptr_goal]++;

          //Then track Harvest Levels and Activity levels by Ownership and periods
          for(c=0;c<NP;c++)
          {
                    if( *ptr_harvest > 0)                                                 //Yes, there was Harvest
activity for this cell in this period
                    {
                              CellsTouched[c]++;                                          //increment counter for
cells touched per period

                              if( *ptr_owner == OWN_USFS || *ptr_owner == OWN_BLM)        //track cells touched &
levels by Fed and NonFed
                              {
                                        FedTouched[c]++;
                                        FedHarvest[c]+= (*ptr_harvest)*ACREEQ;
                              }
                              else
                              {
                                        NonFedTouched[c]++;
                                        NonFedHarvest[c]+= (*ptr_harvest)*ACREEQ;
                              }
                    }

                    ptr_harvest++;                          //increment to next period
          }

}//end for(r=0; r<UNIQUE;r++)


#ifdef DEBUG_OUT_ACRES_HARVEST
//print out GoalCells
for(r=0;r<GOALS;r++)
          printf("There are %lu cells with goal %d\n",GoalCells[r],r);
printf("and %lu cells that were Non-Forest\n",NonForestCells);


printf("\nThere are a total of %.2lf acres in this simulation\n",TotalCellCount*ACREEQ);

TotalFedHarvest = 0;
```

```
TotalNonFedHarvest = 0;
for(r=0;r<NP;r++)
{
          printf("Period %d:\tAcres of FedTouched = %.2lf and Harvest   %.2lf, \tAcres of NonFedTouched = %.2lf and
Harvest %.2lf\n",
                    r+1, FedTouched[r]*ACREEQ, FedHarvest[r], NonFedTouched[r]*ACREEQ, NonFedHarvest[r]);

          TotalFedHarvest+= FedHarvest[r];
          TotalNonFedHarvest += NonFedHarvest[r];
}

printf("\nTotal harvest on Federal Land (USFS, BLM) is %.2lf CF, and  NonFederal land is %.2lf\n",TotalFedHarvest,
#endif

                    TotalNonFedHarvest);


//Print out data to use in EXCEL
fprintf(WriteExcel, "AcresFed\tHarvestFed\t\t\tAcresNonFed\tHarvestNonFed\n");
for(r=0;r<NP;r++)
          fprintf(WriteExcel, "%-6.3lf\t\t%-10.3lf    \t\t\t%-6.2lf\t\t%-10.2lf\n",
                    FedTouched[r]*ACREEQ, FedHarvest[r], NonFedTouched[r]*ACREEQ, NonFedHarvest[r]);


//close the files
fclose(WriteExcel);

}//end OutputAcresHarvest



// ************************************************************
void OutputMapGoals(int Status)
// ************************************************************
{

/*The current goal assignment is outputted to ...\ouputs\PreSimData\goal*\goal.asc in this function.
There is also one outputted to ...\ouputs\rerun_data\goal.(.asc or .bin) during optimization routine and
there is no difference except that the one in ..\*\rerun_data\ is always made in binary format.
*/


//Variable for writing the output files
FILE *WriteGoal;
char GoalFile[256];


int *ptr_srp;                 //Starting Row Position
ushort   *ptr_column, *ptr_goal;
int r,c,HowMany;
int ColumnsLeft, ctr;
ushort StartColumn,OutColumn;
//--------------------------------------- End of variable defining -------------------------------------------

//Make some  filename and open
if(Status == PREDICTED)
          sprintf(GoalFile, "%s%s%d\\goal.asc",PREFIX,PreSimOutputDir,GOAL_TO_USE);
else
          sprintf(GoalFile, "%s%s%d\\goal.asc",PREFIX,PostSimOutputDir,GOAL_TO_USE);

WriteGoal =                   fopen(GoalFile, "w");
if (WriteGoal == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", GoalFile, strerror(errno));



//Print out an ascii file that is in row/column format which contains the GOAL values for every cell
//This file can be used in ArcInfo to make maps!

          //write out the header data
          fprintf(WriteGoal,"ncols\t\t%d\n",COLUMNS);
          fprintf(WriteGoal,"nrows\t\t%d\n",ROWS);
          fprintf(WriteGoal,"xllcorner\t%.6lf\n",F_XLL);
          fprintf(WriteGoal,"yllcorner\t%.6lf\n",F_YLL);
          fprintf(WriteGoal,"cellsize\t%d\n",CELLSIZE);
          fprintf(WriteGoal,"NODATA_value\t%d\n",NODATA);

                    for(r=1;r<=ROWS;r++)
                    {
                              ptr_srp = &link[r-1][1];
                              HowMany = *(ptr_srp+1);
                              StartColumn = Data.GridColumn[(*ptr_srp)-1];
                              ptr_column = &Data.GridColumn[(*ptr_srp)-1];
                              ptr_goal = &Data.Goal[(*ptr_srp)-1];

                              //If the whole row is blank, print out NODATA and goto next row
                              if( *ptr_srp == FALSE )                  //means a zero was left in this spot during
MakeLink
                              {
                                        for(c=1;c<=COLUMNS;c++)
                                                  fprintf(WriteGoal,"%d  ",NODATA);

                                        //put in new lines
                                        fprintf(WriteGoal, "\n");
```

```
                    continue;           //goto next row
                }

                //print out NODATA for those cells before data starts
                for(c=1;c<StartColumn;c++)
                    fprintf(WriteGoal,"%d  ",NODATA);


                //set some counters
                OutColumn = StartColumn;
                ctr = 0;

                //print out values for area on landscape by checking
                //value in Data.GridColumn to match it with OutColumn value
                do{
                    if(*ptr_column == OutColumn)
                    {
                        fprintf(WriteGoal, "%hu  ",*ptr_goal);


                        ptr_goal++;
                        ptr_column++;
                        OutColumn++;
                        ctr++;
                    }
                    else      //print out NODATA for the "gaps"
                    {
                        fprintf(WriteGoal,"%d  ",NODATA);

                        OutColumn++;
                    }
                }while(ctr != HowMany );

                //Check to see how many columns are left to do
                ColumnsLeft = COLUMNS - (OutColumn-1);

                if(ColumnsLeft == 0)
                {
                    fprintf(WriteGoal, "\n");

                    continue;                       //go to next row
                }

                //print out NODATA for those cells after the data that are left
                for(c=0;c<ColumnsLeft;c++)
                        fprintf(WriteGoal,"%d  ",NODATA);

                //put in a new line
                    fprintf(WriteGoal, "\n");

            }//end of for(r=1;r<=ROWS;r++)

fclose(WriteGoal);
}//end MapGoals

// ************************************************************
void OutputForestDistribution(int Status)
// ************************************************************
{
//Variable for writing the output files
FILE *WRITE_VEG, *WRITE_STAGE, *WRITE_COMBO;
char VegDistFile[150], StageDistFile[150], ComboFile[150];

//Acre counters (number of cells)
ulong NonForestCells, TotalCellCount,PerTotal;

//Arrays to hold # of cells for various combination
ulong EntireVeg[NP][VEGCLASSES+1],
AllFedVeg[NP][VEGCLASSES+1],FedNMVeg[NP][VEGCLASSES+1],AllNonFedVeg[NP][VEGCLASSES+1];
ulong EntireStage[NP][STAGES], AllFedStage[NP][STAGES],FedNMStage[NP][STAGES],AllNonFedStage[NP][STAGES];
ulong Combo[STAGES][VEGCLASSES+1][NP];

int r,c,t;
ulong *ptr_cellid, *ptr_treelist;
ushort  *ptr_owner, *ptr_buffer, *ptr_vegcode, *ptr_veg, *ptr_stage, *ptr_alloc;
ushort TempCode;
int VegCode, StageCode, TempVeg, TempDiam, TempCover;
//=====================================end variables ============================================================
/*
Here's the conversion.  Values in Data.Vegcode are those 3 or 4 digits values that were either generated directly
in PREMO or were slightly modified by this progrom in FillPremoData().  Heidi gave me the following regarding
what the PREMO codes meant:

1st digit = (veg. class)
1         CH
2         DH
3         EH
4         CCP
5         MC
6         open ????
7         Pine
8         RF
9         WF

2nd digit = (QMD)
```

```
0         0-4.9
1         5-8.9
2         9-14.9
3         15-20.9
4         21-24.9
5         25-31.9
6         32+

3rd digit = (Canopy closure)
0         <= 60%
1         > 60%

Alterations:

FillInitialPremoData() changed those with an original 1st digit of 5 to be either 5 (MC < 3000') or 10 (MC >
10000'),
so I can directly check for 5 or 10.

The digit assignment from PREMO is not consistent with the already established values
I use for maps in ArcInfo and other tracking so I will use the following conversion matrix:

GIS codes 1-4 are for:  Barren, Water, Shrub, Grass/Forbs respectivly.  Either Vegetation or Seral Stage.  These
were
considered NONFOREST cells in the simulation and should have both a NONFOREST flag in Data.Vegcode and
Data.Treelist.
However, the original classification (barren, water, shrub, grass/forbs) was kept in Data.InitialVeg &
Data.InitialStage (1-4).


(note: this is for the 1st digit(s) only = VEGETATION CLASS )
PREMO               'meaning'           GIS VEGETATION code (this is what I will use to place in correct array
position)
-------------------------------------------------------------------------------------------------------------
1                           CH                                  11
2                           DH                                  10
3                           EH                                  12
4                           CCP                                 9         //PREMO appears not to be
classifying anything as 4** so don't worry if none seen
5                           MC<3000'            6
6                           'open'                              14        //This was not part of original
classification - will have to eventually decide what it is!
7                           Pine                8
8                           RF                                  5
9                           WF                                  7
10                          MC>3000'            13

(note:  this is for the 2nd & 3rd digit (or 4-5 if 1st was a 10) only = SERAL STAGE )
PREMO     PREMO             'meaning'
2nd               3rd                       DBH                 Canopy              GIS SERAL STAGE code
(this is what I will use to place in correct array position)
-------------------------------------------------------------------------------------------------------------
--
0                 0 or 1            0-4.9"              any                             5
1                 0                                     5-8.9"            <=60%         6
1                 1                                     5-8.9"            >60%          7
2                 0                                     9-14.9"           <=60%         8
2                 1                                     9-14.9"           >60%          9
3                 0                                     15-20.9"  <=60%            10
3                 1                                     15-20.9"  >60%            11
4                 0                                     21-24.9"  <=60%            12
4                 1                                     21-24.9"  >60%            13
5                 0 or 1            25-31.9"  any                               14
6                 0 or 1            32+"                any                       15
*/

//NOTE = change these to use memset!

//Initialize the arrays
for(r=0;r<NP;r++)                                       //The arrays to hold Vegetation data
{
          for(c=0;c<VEGCLASSES+1;c++)
          {
                    EntireVeg[r][c] = 0;
                    AllFedVeg[r][c] = 0;
                    FedNMVeg[r][c] = 0;
                    AllNonFedVeg[r][c] = 0;
          }
}
for(r=0;r<NP;r++)                                       //The arrays to hold Seral Stage data
{
          for(c=0;c<STAGES;c++)
          {
                    EntireStage[r][c] = 0;
                    AllFedStage[r][c] = 0;
                    FedNMStage[r][c] = 0;
                    AllNonFedStage[r][c] = 0;
          }
}
for(r=0;r<STAGES;r++)                                           //The array to hold the VegStageCombo data
{
          for(c=0;c<VEGCLASSES+1;c++)
          {
                    for(t=0;t<NP;t++)
                    {
                              Combo[r][c][t] = 0;
```

```
}}}

//Start at beginning of Data.*[] arrays and keep tally of items to output.
NonForestCells=0;
TotalCellCount=0;
for(r=0; r<UNIQUE;r++)
{
          //set pointers
          ptr_cellid =        &Data.Cellid[r];
          ptr_treelist =      &Data.Treelist[r];
          ptr_owner =                 &Data.Owner[r];
          ptr_alloc =                 &Data.Alloc[r];
          ptr_buffer =        &Data.Buffer[r];
          ptr_vegcode =       &Data.Vegcode[r][0];
          ptr_veg =           &Data.InitialVeg[r];
          ptr_stage =                 &Data.InitialStage[r];

          if( *ptr_cellid == FALSE)                                   //no more records to check in array
                    break;

          TotalCellCount++;

          //Check the cells treelist, if NONFOREST, then its 'ptr_vegcode' should be NONFOREST as well, so use
initial Veg & Stage
          if( *ptr_treelist == NONFOREST)
          {
                    NonForestCells++;                                           //keep track of these

                    if( *ptr_vegcode != NONFOREST)                    //problem - this should be NONFOREST
                              Bailout(63);

                    // **** otherwise, look at ptr_veg & ptr_stage and track those values by correct ownership
category ***
                    //Since all arrays were initialized with zero's, I will just increment up a 'hit' which can
then be counted for acres

                    //For the EntireStage and EntireVeg arrays        - This gets filled no matter what
                    for(c=0;c<NP;c++)
                    {
                              EntireVeg[c][(*ptr_veg)-1]++;             //*ptr_veg should have its original veg -
subtract 1 to get array notation
                              EntireStage[c][(*ptr_stage)-1]++;         //*ptr_stage should have its original stage-
subtract 1 to get array notation

                              Combo[(*ptr_stage)-1][(*ptr_veg)-1][c]++;         //Track the intersection of these
for each period
                    }//end for(c=0;c<NP;c++)


                    //For the AllFed* arrays  && the FedNM* array
                    if( *ptr_owner == OWN_BLM || *ptr_owner == OWN_USFS)
                    {
                              for(c=0;c<NP;c++)                                                                   //All
Federal ownership
                              {
                                        AllFedVeg[c][(*ptr_veg)-1]++;
                                        AllFedStage[c][(*ptr_stage)-1]++;
                              }

                              //NOTE: The stream buffer behavior is a bit wierd.  Data.Alloc does NOT have a code
to indicate whether
                              //a cell is in a riparian reserve, that data is in Data.Buffer.  So make sure to look
at Data.Buffer !
                              if(*ptr_alloc == ALLOC_RESERVE || *ptr_alloc == ALLOC_WILD || *ptr_buffer ==
IN_BUFFER)          //LSR, Wilderness, & Riparian
                              {
                                        for(c=0;c<NP;c++)
                                        {
                                                  FedNMVeg[c][(*ptr_veg)-1]++;
                                                  FedNMStage[c][(*ptr_stage)-1]++;
                                        }
                              }
                    }

                    //For the NON-FEDERAL lands, AllNonFed*[][]
                    if( *ptr_owner != OWN_BLM && *ptr_owner != OWN_USFS )       //these are NonFederal lands
                    {
                              for(c=0;c<NP;c++)
                              {
                                        AllNonFedVeg[c][(*ptr_veg)-1]++;
                                        AllNonFedStage[c][(*ptr_stage)-1]++;
                              }
                    }

          }//end if( *ptr_treelist == NONFOREST)

          else                                                //For all NonForest cells, convert the values in
Data.Vegcode[][] and track by same categories
          {

                    //first, extract each periods vegcode and break it apart to get the correct GIS Veg and Stage
values
                    for(c=0;c<NP;c++)
                    {
```

```
                        TempCode = Data.Vegcode[r][c];                          //The actual 3 or 4 digit
code from PREMO

                        //extract the digits out
                        TempCover = TempCode%10;
        //last digit for determining stage (is closure, <=60% or > 60% )
                        TempDiam = ( (TempCode-TempCover)%100 ) / 10;     //next to last digit also for
determining stage (is the QMD group)
                        TempVeg = (TempCode-TempCode%100) / 100;                //1st or 1st two digits
for determining VegCode

                        //Use TempVeg to determine proper GIS VegCode
                        switch(TempVeg)
                        {
                        case 1:             VegCode = 11;       break;          //CH
                        case 2:             VegCode = 10;       break;          //DH
                        case 3:             VegCode = 12;       break;          //EH
                        case 4:             VegCode =  9;       break;          //CCP
                        case 5:             VegCode =  6;       break;          //MC < 3000'
                        case 6:             VegCode = 14;       break;          // 'open'
                        case 7:             VegCode =  8;       break;          //pine
                        case 8:             VegCode =  5;       break;          //RF
                        case 9:             VegCode =  7;       break;          //WF
                        case 10:  VegCode = 13;       break;            //MC > 3000'

                        default:  Bailout(64);                                  //This will exit the
program with proper error message
                        }


                        //Use TempDiam and TempCover to determine proper StageCode
                        switch(TempDiam)
                        {
                        case 0:   StageCode = 5;                        break;          //0-4.9   "
        any closure
                        case 1: if(TempCover == 0)                              //5-8.9"
                                                StageCode = 6;
        //              <=60%
                                        else
                                                StageCode = 7;
        //              >60%

        break;
                        case 2: if(TempCover == 0)                              //9-14.9"
                                                StageCode = 8;
        //              <=60%
                                        else
                                                StageCode = 9;
        //              >60%

        break;
                        case 3: if(TempCover == 0)                              //15-20.9"
                                                StageCode = 10;
        //              <=60%
                                        else
                                                StageCode = 11;
        //              >60%

        break;
                        case 4: if(TempCover == 0)                              //21-24.9"
                                                StageCode = 12;
        //              <=60%
                                        else
                                                StageCode = 13;
        //              >60%

        break;
                        case 5: StageCode = 14;                         break;          //25-31.9"
        any closure
                        case 6: StageCode = 15;                         break;          //32+'
        any closure

                        default:  Bailout(65);                                  //This will
exit the program with proper error message
                        }

        //********** Now fill the appropriate cell tracking arrays based on ownership breakdown **********

                        //For the EntireStage and EntireVeg arrays        - These get filled no matter what
                        EntireVeg[c][VegCode-1]++;
        //subtract 1 to get array notation
                        EntireStage[c][StageCode-1]++;

                        Combo[StageCode-1][VegCode-1][c]++;

                        //For the AllFed* arrays  && the FedNM* array
                        if( *ptr_owner == OWN_BLM || *ptr_owner == OWN_USFS)                    //All
Federal ownership
                        {
                                        AllFedVeg[c][VegCode-1]++;
                                        AllFedStage[c][StageCode-1]++;

                                if(*ptr_alloc == ALLOC_RESERVE || *ptr_alloc == ALLOC_WILD || *ptr_buffer
== IN_BUFFER}       //LSR,Riparian,& wilderness
                                {
```

```
                                                FedNMVeg[c][VegCode-1]++;
                                                FedNMStage[c][StageCode-1]++;
                                        }
                                }

                                //For the NON-FEDERAL lands, AllNonFed*[][]
                        if( *ptr_owner != OWN_BLM && *ptr_owner != OWN_USFS )       //these are NonFederal lands
                        {
                                                AllNonFedVeg[c][VegCode-1]++;
                                                AllNonFedStage[c][StageCode-1]++;
                                }

                        }//end for(c=0;c<NP;c++)
                }//end else if( *ptr_treelist == NONFOREST)

}//end for(r=0; r<UNIQUE;r++)


//printf("Checked all the cells during OutputForestDistribution\n");

// ************************** Print out the data ******************************************
//Make some output filenames and open files

if(Status == PREDICTED)                                                 //This is PreSimulation data
{
        sprintf(VegDistFile, "%s%s%d\\VegDist.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
        sprintf(StageDistFile, "%s%s%d\\StageDist.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
        sprintf(ComboFile, "%s%s%d\\ComboDist.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
}
else                                                                                    //This is PostSimulation
data
{
        sprintf(VegDistFile, "%s%s%d\\VegDist.txt",PREFIX,PostSimOutputDir,GOAL_TO_USE);
        sprintf(StageDistFile, "%s%s%d\\StageDist.txt",PREFIX,PostSimOutputDir,GOAL_TO_USE);
        sprintf(ComboFile, "%s%s%d\\ComboDist.txt",PREFIX,PostSimOutputDir,GOAL_TO_USE);
}


WRITE_VEG =         fopen(VegDistFile, "w");
WRITE_STAGE= fopen(StageDistFile, "w");
WRITE_COMBO = fopen(ComboFile, "w");
if (WRITE_VEG == NULL)
        fprintf(stderr, "opening of %s failed: %s\n", VegDistFile, strerror(errno));
if (WRITE_STAGE == NULL)
        fprintf(stderr, "opening of %s failed: %s\n", StageDistFile, strerror(errno));
if (WRITE_COMBO == NULL)
        fprintf(stderr, "opening of %s failed: %s\n", ComboFile, strerror(errno));



//Put out the combo file first, it will have a Stage-Veg acre matrix for each period

for(c=0;c<NP;c++)   //For each period
{
fprintf(WRITE_COMBO, "Per%d\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t14\n",c+1);
fprintf(WRITE_COMBO, "----------------------------------------------------------------------------------------------------
----------\n");
for(r=0;r<STAGES;r++)
{
        fprintf(WRITE_COMBO, "%d\t",r+1);
        for(t=0;t<VEGCLASSES+1;t++)
        {
                fprintf(WRITE_COMBO, "%-6.0f\t",Combo[r][t][c]*ACREEQ);
        }
        fprintf(WRITE_COMBO, "\n");
}
fprintf(WRITE_COMBO, "\n\n\n");
}



//Put in some header lines for the Entire* arrays
fprintf(WRITE_VEG, "Entire\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t14\n");
fprintf(WRITE_VEG, "----------------------------------------------------------------------------------------------------
----------\n");
fprintf(WRITE_STAGE, "Entire\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t14\t15\n");
fprintf(WRITE_STAGE, "----------------------------------------------------------------------------------------------------
----------\n");
for(r=0;r<NP;r++)
{
        //Put in the period
        fprintf(WRITE_VEG,"%d",r+1);
        fprintf(WRITE_STAGE, "%d",r+1);

        //The Veg acres
        PerTotal=0;
        for(c=0;c<VEGCLASSES+1;c++)
        {
                PerTotal+=EntireVeg[r][c];
                fprintf(WRITE_VEG, "\t%-6.0f",EntireVeg[r][c]*ACREEQ);
        }
                fprintf(WRITE_VEG, "\t\t%-8.2f",PerTotal*ACREEQ);

        //The Stage acres
        PerTotal=0;
```

```
		for(c=0;c<STAGES;c++)
		{
			PerTotal+=EntireStage[r][c];
			fprintf(WRITE_STAGE, "\t%-6.0f",EntireStage[r][c]*ACREEQ);
		}
			fprintf(WRITE_STAGE, "\t\t%-8.2f",PerTotal*ACREEQ);

		//Put in a new line
		fprintf(WRITE_VEG,"\n");
		fprintf(WRITE_STAGE, "\n");
}


//A couple of spaces to separate next array data
fprintf(WRITE_VEG, "\n\n");
fprintf(WRITE_STAGE, "\n\n");

//Put in some header lines for AllFed* arrays
fprintf(WRITE_VEG, "AllFed\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t14\n");
fprintf(WRITE_VEG, "-------------------------------------------------------------------------------------------------------------------
--------------------\n");
fprintf(WRITE_STAGE, "AllFed\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t13\t15\n");
fprintf(WRITE_STAGE, "-------------------------------------------------------------------------------------------------------------------
---------------------------------------\n");
for(r=0;r<NP;r++)
{
		//Put in the period
		fprintf(WRITE_VEG,"%d",r+1);
		fprintf(WRITE_STAGE, "%d",r+1);

		//The Veg acres
		PerTotal=0;
		for(c=0;c<VEGCLASSES+1;c++)
		{
			PerTotal+=AllFedVeg[r][c];
			fprintf(WRITE_VEG, "\t%-6.0f",AllFedVeg[r][c]*ACREEQ);
		}
			fprintf(WRITE_VEG, "\t\t%-8.2f",PerTotal*ACREEQ);

		//The Stage acres
		PerTotal=0;
		for(c=0;c<STAGES;c++)
		{
			PerTotal+=AllFedStage[r][c];
			fprintf(WRITE_STAGE, "\t%-6.0f",AllFedStage[r][c]*ACREEQ);
		}
			fprintf(WRITE_STAGE, "\t\t%-8.2f",PerTotal*ACREEQ);

		//Put in a new line
		fprintf(WRITE_VEG,"\n");
		fprintf(WRITE_STAGE, "\n");
}


//A couple of spaces to separate next array data
fprintf(WRITE_VEG, "\n\n");
fprintf(WRITE_STAGE, "\n\n");

//Put in some header lines for FedNM* arrays
fprintf(WRITE_VEG, "FedNM\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t14\n");
fprintf(WRITE_VEG, "-------------------------------------------------------------------------------------------------------------------
--------------------\n");
fprintf(WRITE_STAGE, "FedNM\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t13\t15\n");
fprintf(WRITE_STAGE, "-------------------------------------------------------------------------------------------------------------------
---------------------------------------\n");
for(r=0;r<NP;r++)
{
		//Put in the period
		fprintf(WRITE_VEG,"%d",r+1);
		fprintf(WRITE_STAGE, "%d",r+1);

		//The Veg acres
		PerTotal=0;
		for(c=0;c<VEGCLASSES+1;c++)
		{
			PerTotal+=FedNMVeg[r][c];
			fprintf(WRITE_VEG, "\t%-6.0f",FedNMVeg[r][c]*ACREEQ);
		}
			fprintf(WRITE_VEG, "\t\t%-8.2f",PerTotal*ACREEQ);
		//The Stage acres
		PerTotal=0;
		for(c=0;c<STAGES;c++)
		{
			PerTotal+=FedNMStage[r][c];
			fprintf(WRITE_STAGE, "\t%-6.0f",FedNMStage[r][c]*ACREEQ);
		}
			fprintf(WRITE_STAGE, "\t\t%-8.2f",PerTotal*ACREEQ);

		//Put in a new line
		fprintf(WRITE_VEG,"\n");
		fprintf(WRITE_STAGE, "\n");
}


//A couple of spaces to separate next array data
```

```
fprintf(WRITE_VEG, "\n\n");
fprintf(WRITE_STAGE, "\n\n");

//Put in some header lines for AllNonFed* arrays
fprintf(WRITE_VEG, "NonFed\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t14\n");
fprintf(WRITE_VEG, "----------------------------------------------------------------------------------------------------
-------------------\n");
fprintf(WRITE_STAGE, "NonFed\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\t13\t15\n");
fprintf(WRITE_STAGE, "----------------------------------------------------------------------------------------------------
-------------------------------------\n");
for(r=0;r<NP;r++)
{
          //Put in the period
          fprintf(WRITE_VEG,"%d",r+1);
          fprintf(WRITE_STAGE, "%d",r+1);

          //The Veg acres
          PerTotal=0;
          for(c=0;c<VEGCLASSES+1;c++)
          {
                    PerTotal+=AllNonFedVeg[r][c];
                    fprintf(WRITE_VEG, "\t%-6.0f",AllNonFedVeg[r][c]*ACREEQ);
          }
                    fprintf(WRITE_VEG, "\t\t%-8.2f",PerTotal*ACREEQ);

          //The Stage acres
          PerTotal=0;
          for(c=0;c<STAGES;c++)
          {
                    PerTotal+=AllNonFedStage[r][c];
                    fprintf(WRITE_STAGE, "\t%-6.0f",AllNonFedStage[r][c]*ACREEQ);
          }
                    fprintf(WRITE_STAGE, "\t\t%-8.2f",PerTotal*ACREEQ);

          //Put in a new line
          fprintf(WRITE_VEG,"\n");
          fprintf(WRITE_STAGE, "\n");
}



fclose(WRITE_VEG);
fclose(WRITE_STAGE);
fclose(WRITE_COMBO);
}//end OutputForestDistribution

// ***************************************************************
void LevelOfActivity(int Status)
// ***************************************************************
{
/* This will output a table with 6th field subwatershed id's in Rows,
and columns for the four EvaluateThisPeriod[] periods, with values representing how many
acres were "touched".  This file will also have its first column (after
the id) with the total acres in that subwatershed and then the total forested acres.
This file will be comma delimited and can be imported into ArcInfo and joined with the
SubWatershed layer to make maps showing the LevelOfActivity - or the
tables can be used stand-alone.

NOTE:  This function is not looking at a "solution" to determine the LOA, but that should be negligible because
this tracks harvest values and a cell not in the solution for a particular landscape goal will not have any
harvest associated with it anyways!
*/

//These globals were filled when main() called up CountSubWatersheds
extern int UniqueMinor[300];
extern int USW;

int a, r, SearchShed;
ushort *ptr_minor;
ulong *ptr_treelist;
int PerA, PerB, PerC, PerD, Hit;

FILE *WRITE_LOA;
char LoaFile[256];
//-------------------------------- End of Variables --------------------------------------------------


printf("There are %d 6th-field subwatersheds in LevelOfActivity\n",USW);

//Create on FreeStore an array to hold rows for each subwatershed and 7 columns:
//                  [][0]=ID  [][1]=TotalCells  [][2]=ForestedCells  [][3]=Cells in Per"A"
//                  [][4]=Cells in Per"B"  [][5]=Cells in Per"C"  [][6]=Cells in Per"D"
ulong (*LOA)[7] = new ulong[USW][7];
if (LOA == NULL)
          printf("There was NOT enough memory for LOA with %lu elements\n", USW*7);

//initialize the LOA array
for(r=0;r<USW;r++)
{
          for(a=0;a<7;a++)
          {
                    LOA[r][a] = 0;
}}

//Look at EvaluateThisPeriod and find the 4 evaluation periods there
```

```
Hit=0;
for(r=0;r<NP;r++)
{
          if(EvaluateThisPeriod[r] > 0)
          {

                    if(Hit == 4)
                    {
                              printf("There are too many EvaluateThisPeriod[] periods! - ignoring those past the
first four\n");
                              break;
                    }

                    if(Hit == 0)
                              PerA = r;
                    else if(Hit == 1)
                              PerB = r;
                    else if(Hit == 2)
                              PerC = r;
                    else
                              PerD = r;

                    Hit++;
          }
}//end for(r=0;r<NP;r++)

//printf("Evaluating Periods:  %d, %d, %d, %d\n",PerA+1, PerB+1, PerC+1, PerD+1);


//Start the search process
for(r=0;r<USW;r++)
{
          SearchShed = UniqueMinor[r];

          if(SearchShed == WATER_BODY || SearchShed == NODATAFLAG)
                    continue;

          //Put ID in LOA
          LOA[r][0] = SearchShed;

          //Using SearchShed, look through all of Data.Minor for that value
          for(a=0;a<UNIQUE;a++)
          {
                    ptr_minor = &Data.Minor[a];
                    ptr_treelist = &Data.Treelist[a];

                    if(*ptr_minor == 0)
                              break;                                        //assumes Data.Minor was initialized with 0's
and there are
                                                                                        //no actual Minor sub-
watershed values of 0

                    if( (ushort)SearchShed == *ptr_minor )                        //YES, they match
                    {

                              //Tally up the total acres for this subwatershed
                              LOA[r][1]++;

                              //Tally up the actual forested acres
                              if( *ptr_treelist != NONFOREST)
                                        LOA[r][2]++;

                              //Look at appropriate Data.CFHarvest elements to see if there was activity or not
                              //If there was, tally up the number of cells as appropriate
                              if( Data.CFHarvest[a][PerA] > 0 )                 //There was a harvest in Period "A"
                                        LOA[r][3]++;

                              if( Data.CFHarvest[a][PerB] > 0 )                 //There was a harvest in Period "B"
                                        LOA[r][4]++;

                              if( Data.CFHarvest[a][PerC] > 0 )                 //There was a harvest in Period "C"
                                        LOA[r][5]++;

                              if( Data.CFHarvest[a][PerD] > 0 )                 //There was a harvest in Period "D"
                                        LOA[r][6]++;
                    }
          }//end for(a=0;a<UNIQUE;a++)
}//end for(r=0;r<USW;r++)


// Create, Open, and Write data out to a file
if(Status == PREDICTED)
                              //This is PreSimulation data
          sprintf(LoaFile, "%s%s%d\\loa.csv",PREFIX,PreSimOutputDir,GOAL_TO_USE);
else
                                        //This is PostSimulation data
          sprintf(LoaFile, "%s%s%d\\loa.csv",PREFIX,PostSimOutputDir,GOAL_TO_USE);

WRITE_LOA =         fopen(LoaFile, "w");
if (WRITE_LOA == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", LoaFile, strerror(errno));



//No header line because ArcInfo won't import them - see top of function for format
```

```
//Will output the actual acres associated with the cell count found -- ** no TABS either (A/I doesn't like them)
for(r=0;r<USW;r++)
{
          if(LOA[r][0] > 0 )
          {
                    fprintf(WRITE_LOA,"%lu,",LOA[r][0]);              //the ID is stored here - don't convert to
acres
                    for(a=1;a<7;a++)
                    {
                              fprintf(WRITE_LOA,"%-7.2f",LOA[r][a]*ACREEQ);
                              if(a < 6 )
          //don't want comma after last value - screws ArcInfo up
                                        fprintf(WRITE_LOA,",");
                    }
                    fprintf(WRITE_LOA,"\n");
          }
}

delete [] LOA;

fclose(WRITE_LOA);
}//end LevelOfActivity

// ********************************************************************************************
void TimingChoiceFrequency(void)
// ********************************************************************************************
{
/*
The objective is to look at all the SD_*_*_*.txt files for ALL the possible prescriptions that could
be chosen for the initial landscape (ALL means for all 10 stand goals and 2 "hold" periods for all existing
treelist!).
This function will count up the total number of prescriptions opened and track, by period, how many prescriptions
had harvesting (i.e. thinnings) occuring in each period.  This frequency can then be compared to the
harvest values that are seen after a landscape optimization and notice if harvest flow is occuring with period
peaks that also have a high frequency of prescriptions with harvesting in that period (even-flow may be
difficult to achieve because of that.
*/

FILE *Index, *SD, *WriteOut;
char Garbage[100]="",Temp[256], SDFile[256];
int ScanStatus,IndexNo,count, ctr, goal, HoldPeriods;
int TotalFiles,y;

int AF[NP];
double TotalVolume[NP];
int DataPeriod;
double RealBasal, RealClosure, RealCBD, RealHLC, RealHeight, RealRev, RealBigTrees, Harvest, SD_Era;
ushort VegCode,RealLitter,RealClass25,RealClass1,RealClass3,RealClass6,RealClass12,RealClassOver12;

//------------------------------------------- End of variable defining ----------------------------------------

//Create the Data.*[] arrays so Data.Treelist gets made for the particular ENVIRONMENT defined
CreateMainData();


//initialize array
for(y=0;y<NP;y++)
{
          AF[y]=0;
          TotalVolume[y] = 0;
}


// I will assume that the treelist index.txt file is completely filled with valid stands and files
          sprintf(Temp, "%s%s%d\\per0\\%s",PREFIX,INPUTS,GOAL_TO_USE,TREE_INDEX);
          Index = fopen(Temp,"r");

          if (Index == NULL)
                    fprintf(stderr, "opening of %s failed:  %s\n", Temp,  strerror(errno));


          // First go through the file and COUNT the number of files
          count = 0;
          while ((ScanStatus=fscanf(Index,"%d",&IndexNo))!=EOF)
          {
                    count = ++count;
          }

          printf("\n\nThere are %d files in the Tree Index File\n\n",count);

          // Rewind the file pointer so it is back at the beginning of the file
          rewind(Index);


//For each treelist-goal-hold possibility, go through read the SD_*_*_.txt file and track periodic harvest
frequency
TotalFiles=0;
for(ctr = 0; ctr < count; ctr++)                                        //for each treelist
{
          fscanf(Index,"%d",&IndexNo);                                  //Scan the index no.

          //flag for NONFOREST values
          if(IndexNo == NONFOREST)
                    continue;
```

```
        for(goal=0;goal<GOALS;goal++)                                                    //for each goal
        {

                //Set a quick error if I change the # of HoldFor periods and I forget to fix this code
                if(HOLDNO > 2)
                        Bailout(40);

                for(HoldPeriods=0;HoldPeriods<4;HoldPeriods+=3)   //for the two Hold "for" periods
                {
                        TotalFiles++;

                        //Make the appropriate file name and actually open the SD_*_*_*.txt file
                        sprintf(SDFile,
"%s%s\\SD_%d_%d_%d.txt",PREFIX,InitialStandDataDir,IndexNo,goal,HoldPeriods);
                        SD = fopen(SDFile,"r");
                        if (SD == NULL)
                                fprintf(stderr, "opening of %s failed:  %s\n", SDFile, strerror(errno));


                        //First, scan in the the first line from the SD* file-which is for Time 0, do not
want
                        fscanf(SD,"%d %lf %lf %lf %lf %lf %lf %lf %hu %lf %lf %lf %lf %lf %lf %lf",
&DataPeriod, &RealBasal,
                                &RealClosure, &RealCBD, &RealHLC, &RealHeight, &RealRev, &RealBigTrees,
&SD_Era, &VegCode, &Harvest,
                                &RealLitter, &RealClass25, &RealClass1, &RealClass3, &RealClass6,
&RealClass12, &RealClassOver12);

                        for(y=1;y<=NP;y++)
                        {
                                //Now actually scan in the data for all the modeling periods
                                fscanf(SD,"%d %lf %lf %lf %lf %lf %lf %lf %lf %hu %lf %lf %lf %lf %lf %lf
%lf %lf",   &DataPeriod, &RealBasal,
                                        &RealClosure, &RealCBD, &RealHLC, &RealHeight, &RealRev,
&RealBigTrees, &SD_Era, &VegCode, &Harvest,
                                        &RealLitter, &RealClass25, &RealClass1, &RealClass3, &RealClass6,
&RealClass12, &RealClassOver12);

                                if(Harvest > 0 && y < HoldPeriods)
                                        printf("Prescription P_%d_%d_%d.txt has harvest occuring before
HoldPeriod expires!\n",IndexNo,goal,HoldPeriods);

                                if(DataPeriod != y)
                                        printf("PROBLEM - there aren't NP periods in file %s\n",SDFile);

                                //If there is a value > 0 for Harvest, increment the AF[y] array by one
                                //And track the total volume
                                if(Harvest > 0 )
                                {
                                        AF[y-1]++;
                                        TotalVolume[y-1] += Harvest;
                                }
                        }//end for(y=1;y<=NP;y++)

                        fclose(SD);
                }
        }
}//end for(ctr = 0; ctr < count; ctr++)


fclose(Index);

//print out the results to screen
printf("================== Harvest Frequency Analysis ====== %d files================\n",TotalFiles);
puts("");
for(y=0;y<NP;y++)
        printf( "Period %d:\t%d\t\t%.0f\t\t%.3lf\n",y+1,AF[y],(float) AF[y]/TotalFiles*100,TotalVolume[y]/AF[y]
);

//and print out results to a file
sprintf(Temp,"%s%s\\HarvestFrequency.txt",PREFIX,GeneralDataDir);
WriteOut = fopen(Temp, "w");                                                                    //no
error checking
fprintf(WriteOut,"================== Harvest Frequency Analysis ====== %d files==============\n\n",TotalFiles);
fprintf(WriteOut, "\t\tNo. of \t\tPercent of\tAvg.Volume\n");
fprintf(WriteOut, "\t  Prescriptions\t\t Total\t\tper prescription\n");
fprintf(WriteOut,"\n");
for(y=0;y<NP;y++)
        fprintf(WriteOut,"Period %d:\t%d\t\t%.0f\t\t%.3lf\n",y+1,AF[y],(float)
AF[y]/TotalFiles*100,TotalVolume[y]/AF[y] );

fclose(WriteOut);

}//end TimingChoiceFrequency


// ********************************************************************************************
void OwnershipByMinor(int USW, int UniqueMinor[])
// ********************************************************************************************
{                                       //Figure out the majority owner for each subwatershed
//NOTE:  This is not a "perfect" method and is suited to be changed as seen fit

FILE *WriteOut;
char Temp[256];
int Shed,CurrentShed;
```

```
int r;

//-------------------------------- End of variable defining ----------------------------------------


//Make an array on free store that will store, for each subwatershed, the # of cells by ownership category
//rows: subwatershed      columns: [0],subwatershed #    [1],Federal(BLM, USFS)      [2],All Others
ulong (*MO)[3] = new ulong[USW][3];
                                        //MinorOwner (MO)
if (MO == NULL)
          printf("There was NOT enough memory for MO with %lu elements\n", USW*2);

//initialize the array
for(r=0;r<USW;r++)
{
          MO[r][0]=0;
          MO[r][1]=0;
          MO[r][2]=0;
}

//Start to look at each sub-watershed, one at a time, and track ownership
for(Shed=0;Shed<USW;Shed++)
{
          //get the appropriate sub-watershed value from the UniqueMinor array
          CurrentShed=UniqueMinor[Shed];

          MO[Shed][0] = (ulong)CurrentShed;                              //populate the Subwatershed #

          //Start looking through Data.* arrays and find subwatersheds with this value and track ownership
          for(r=0;r<UNIQUE;r++)
          {
                    if(Data.Minor[r] == FALSE)
                              break;

                    if(Data.Minor[r] == NODATAFLAG)                                   //Some of those GIS slivers or bad
data
                              continue;

                    if(Data.Minor[r] == WATER_BODY)                                   //These are lakes, etc.
                              continue;

                    if(Data.Treelist[r] == NONFOREST)                 //don't count those that are nonforest
anyways
                              continue;

                    //Now make a switch according to which goal is being evaluated and make sure to evaluate only
                    //those cells that are elegible for that goal anyway
                    switch(GOAL_TO_USE)
                    {
                    case 1:
                              if( Data.Alloc[r] == ALLOC_WILD || (Data.Alloc[r] == ALLOC_RESERVE &&
Data.InitialStage[r] > 9) ||
                                        (Data.Buffer[r] == IN_BUFFER && Data.InitialStage[r] > 9) )
                                        continue;
                              break;
                    case 2:
                              if(Data.Alloc[r] == ALLOC_WILD)
                                        continue;
                              break;
                    default:
                              break;
                    )



                    //Only gets to here if all above have passed and no continue statement was encountered
                    if(Data.Minor[r] == (ushort)CurrentShed)
                    {
                              if(Data.Owner[r] == OWN_BLM || Data.Owner[r] == OWN_USFS)
                                        MO[Shed][1]++;
                              else
                                        MO[Shed][2]++;
                    }
          }//end for(r=0;r<UNIQUE;r++)
}//end for(Shed=0;Shed<USW;Shed++)


//print results
printf("Subwatershed #\tMajority Owner\n");
printf("=============================================\n");
for(r=0;r<USW;r++)
{
          printf("%lu :",MO[r][0]);

          if(MO[r][1] > MO[r][2])
                    printf("\t\tFEDERAL\n");
          else if(MO[r][1] < MO[r][2])
                    printf("\t\tNONfederal\n");
          else
                    printf("\t\tequal\n");
}


//and print out results to a file
sprintf(Temp,"%s%s\\Goal%d_OwnerMinor.txt",PREFIX,GeneralDataDir,GOAL_TO_USE);
```

```
WriteOut = fopen(Temp, "w");                                                                    //no
error checking
fprintf(WriteOut, "Subwatershed #\tMajority Owner\n");
fprintf(WriteOut, "=========================================\n");
for(r=0;r<USW;r++)
{
          fprintf(WriteOut, "%lu :",MO[r][0]);

          if(MO[r][1] > MO[r][2])
                    fprintf(WriteOut, "\t\tFEDERAL\n");
          else if(MO[r][1] < MO[r][2])
                    fprintf(WriteOut, "\t\tNONfederal\n");
          else
                    fprintf(WriteOut, "\t\tequal\n");
}


fclose(WriteOut);


delete [] MO;

}//end OwnershipByMinor


// ***********************************************************************************************
void OutputPotentialBigTreesAllStandGoals(void)
// ***********************************************************************************************
{
/*
Output the number of BigTrees associated with the eligible solution area for any one goal applied
as the only solution.  That is, pretend that only one goal is selected, and one hold value, and
call that the "solution" and then count up the Big Trees.

This function will simply use the same rules used to define the Solution cells for a particular goal
- and then use those cells to look up their associated big trees from the PREMO data.
*/


FILE *BinIn, *HeaderIn, *WriteOut;
char Temp[256];
ulong Records;

int a, b,x;
ulong c;
ulong AllocOK, AllocNOK,CellsInShed;
ulong SolutionCounters[3];                                  //will get filled with  AllocOK,  AllocNOK,
CellsInShed, by DetermineEligibleCells()

double PerBigTrees[NP];
double SumBigTrees = 0;

struct OPTIMIZE_SINGLE_VALUE Key;
struct OPTIMIZE_SINGLE_VALUE *ptr_key;
//------------------------------------ End variable defining ---------------------------------------------

//Create the output file for the data generated here
sprintf(Temp, "%s%s%d\\All_BigTrees.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);

//Open up the file for printing
WriteOut = fopen(Temp, "w");


//Open the Header and actual Binary file containing the data found during FillValueToOptimize()
sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.bin",PREFIX,InitialStandDataDir,ENVT);
BinIn = fopen(Temp, "rb");

sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderIn = fopen(Temp, "r");

//Get the Number of records that are listed in the header file
fscanf(HeaderIn,"%lu",&Records);

//Create an array of structures on the free store to hold these records
struct OPTIMIZE_SINGLE_VALUE (*OptValues) = new struct OPTIMIZE_SINGLE_VALUE[Records];
if( OptValues == NULL )
          printf("Problems allocating memory for OptValues[] with %lu
elements\n",Records*sizeof(OPTIMIZE_SINGLE_VALUE));


//Now just read in the binary data the same way it was written out in FillValueToOptimize()
fread(OptValues, sizeof(OPTIMIZE_SINGLE_VALUE),Records,BinIn);

//close up the files
fclose(BinIn);
fclose(HeaderIn);


//Initialize the SolutionCounters array and call up the DetermineEligibleCells() function to fill it up
for(a=0;a<3;a++)
          SolutionCounters[a] = 0;

printf("*** Going to determine the eligible cells for this solution and fill up the array of SOLUTION structures
***\n");
```

```
if( DetermineEligibleCells(SolutionCounters) == FALSE)
          Bailout(82);

//The values now in SolutionCounters should be properly set
AllocOK             = SolutionCounters[0];
AllocNOK  = SolutionCounters[1];
CellsInShed         = SolutionCounters[2];

printf("!!! There are %lu valid cells with cellids.....",CellsInShed);
printf(" and %lu cells that are eligible for the solution and %lu that are not.\n\n",AllocOK,AllocNOK);

//Set a checker to look for when there are 0 eligible cells
if(AllocOK == FALSE)
          Bailout(89);

//Create an array of structures on the free store to hold the solution
struct SOLUTION (*Solution) = new struct SOLUTION[AllocOK];
if( Solution == NULL )
          printf("Problems allocating memory for Solution[] with %lu elements\n",AllocOK*sizeof(SOLUTION));

//Initialize
memset( Solution, 0, sizeof(struct SOLUTION) * AllocOK );

//Now fill that array of SOLUTION structures with the Treelist - Minor - Cellid - GOAL - and HOLD of those eligible
cells
if( FillSolution(SolutionCounters, Solution, REAL) == FALSE )
          Bailout(83);

for(a=0;a<(signed)AllocOK;a++)
          Solution[a].Hold = 0;                                          //assign a Hold value of 0 to all cells

/*****************************************************************************************************************
***
All the above stuff only needs to be done once.  At this point the Solution structure is filled up with the
Treelist-Cellid-Minor values for all the eligible cells in this solution.  To simulate the ideal of applying
one goal across the landscape, just make a loop to fill all of the Solution.Goal[]  members with one goal value.
Then use that as the Key when searching the above OptValues[] structure and copy what was done in the
OutputBigTreesForSolution() function to output the big tree values
*****************************************************************************************************************
****/

//MAKE a loop to do this for each of the StandGoals possible
for(x=0;x<GOALS;x++)
{

          //First, assign goal "x" to all the cells in the Solution structure
          for(a=0;a<(signed)AllocOK;a++)
                    Solution[a].Goal = (ushort)x;


          //Re-initialize the PerBigTrees[] array
          for(a=0;a<NP;a++)
                    PerBigTrees[a] = 0;

          //Also reset SumBigTrees
          SumBigTrees=0;

          for(c=0;c<AllocOK;c++)        //AllocOK is how many rows of data there are (i.e. eligible cells found
earlier)
          {
                    //Make a Key using the Treelist-Goal-Hold values found for each record in the array of Solution
structures
                    Key.Treelist        = Solution[c].Treelist;
                    Key.Goal            = Solution[c].Goal;                    //will all have the value of "x"
                    Key.Hold            = Solution[c].Hold;                    //will all be 0

                    //Now use bsearch to find the matching record in the array of OV structures
                    ptr_key = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
                              &Key,
                              (void *)OptValues,
                              (size_t)Records,
                              sizeof( struct OPTIMIZE_SINGLE_VALUE),
                              LookAtOSV );

                    if(ptr_key == NULL)                                    //There had better be one!
                    {
                              printf("Can't find key number %lu: Treelist = %lu, Goal = %hu, and Hold =
%hu\n",c,Key.Treelist,Key.Goal,Key.Hold);
                              Bailout(80);
                    }
                    else                                                                  //Sum up the periodic
Values
                    {
                              for(b=0;b<NP;b++)
                                        PerBigTrees[b] += ptr_key->BigTrees[b];
                    }
          }//end for(c ...)



          //******** When outputting the # of Big Trees, remember that data was entered by multiplying by 10 - so
divide to get real value
          //Add up the total sum of big trees

          for(b=0;b<NP;b++)
```

```
                    SumBigTrees += PerBigTrees[b]/BIGTREES_EXP;

          fprintf(WriteOut,"\n\nSTAND GOAL: %d\n",x);
          fprintf(WriteOut,"These are Big Trees that were in the Solution area only...which amounted to %.2lf
acres\n",AllocOK*ACREEQ);
          fprintf(WriteOut,"\nThe Period Big Trees Totals are:\n");
          for(a=0;a<NP;a++)
                    fprintf(WriteOut,"Per%d is %-.3lf\n",a+1,PerBigTrees[a]/BIGTREES_EXP);
          fprintf(WriteOut,"The total sum of Big Trees is:  %.3lf\n",SumBigTrees);
          fprintf(WriteOut,"Which amounts to about %.3lf per acre\n",SumBigTrees/(AllocOK*ACREEQ));
}//end for(x=0 ... )


fclose(WriteOut);

//delete stuff on free store
delete [] Solution;
delete [] OptValues;

}//end OutputPotentialBigTreesAllGoals

// ************************************************************
void OutputVegcodes(int Per)
// ************************************************************
{
//NOTE:  the incoming "Per" is the correct period to which this data goes (not array subscript)

//Output the GIS vegcode variable so they can be pulled into a GIS and mapped

FILE *WRITE_VEG;
char VegcodeFile[256];

int *ptr_srp;                  //Starting Row Position
ushort   *ptr_column;
int r,c,HowMany;
int ColumnsLeft, ctr;
ushort StartColumn,OutColumn;
ushort  *ptr_vegcode;

//=====================================end variables ==========================================================

//Make the correct output file name and open it
sprintf(VegcodeFile, "%s%s%d\\per%d\\vegcode.asc",PREFIX,OUTPUTS,GOAL_TO_USE,Per);
WRITE_VEG =                    fopen(VegcodeFile, "w");
if (WRITE_VEG == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", VegcodeFile, strerror(errno));


//Start writting data to the file

fprintf(WRITE_VEG,"ncols\t\t%d\n",COLUMNS);
fprintf(WRITE_VEG,"nrows\t\t%d\n",ROWS);
fprintf(WRITE_VEG,"xllcorner\t%.6lf\n",P_XLL);
fprintf(WRITE_VEG,"yllcorner\t%.6lf\n",P_YLL);
fprintf(WRITE_VEG,"cellsize\t%d\n",CELLSIZE);
fprintf(WRITE_VEG,"NODATA_value\t%d\n",NODATA);

for(r=1;r<=ROWS;r++)
{
          ptr_srp = &link[r-1][1];
          HowMany = *(ptr_srp+1);
          StartColumn =                Data.GridColumn[(*ptr_srp)-1];                    //not a pointer!
          ptr_column =                 &Data.GridColumn[(*ptr_srp)-1];
          ptr_vegcode =                &Data.Vegcode[(*ptr_srp)-1][Per-1];

          //If the whole row is blank, print out NODATA and goto next row
          if( *ptr_srp == FALSE )                 //means a zero was left in this spot during MakeLink
          {
                    for(c=1;c<=COLUMNS;c++)
                              fprintf(WRITE_VEG,"%d  ",NODATA);

                    //put in new line
                    fprintf(WRITE_VEG,"\n");

                    continue;           //goto next row
          }


          //print out NODATA for those cells before data starts
          for(c=1;c<StartColumn;c++)
                    fprintf(WRITE_VEG,"%d  ",NODATA);

          //set some counters
          OutColumn = StartColumn;
          ctr = 0;

          //print out values for area on landscape by checking
          //value in Data.GridColumn to match it with OutColumn value
          do{
                    if(*ptr_column == OutColumn)
                    {
                              fprintf(WRITE_VEG, "%hu  ",*ptr_vegcode);

                              ptr_vegcode+=NP;
```

```
                        ptr_column++;
                        OutColumn++;
                        ctr++;
                }
                else    //print out NODATA for the "gaps"
                {
                        fprintf(WRITE_VEG,"%d  ",NODATA);

                        OutColumn++;
                }
        }while(ctr != HowMany );

        //Check to see how many columns are left to do
        ColumnsLeft = COLUMNS - (OutColumn-1);

        if(ColumnsLeft == 0)
        {
                fprintf(WRITE_VEG, "\n");

                continue;                       //go to next row
        }

        //print out NODATA for those cells after the data that are left
        for(c=0;c<ColumnsLeft;c++)
                fprintf(WRITE_VEG,"%d  ",NODATA);


        //put in a new line
        fprintf(WRITE_VEG, "\n");

}//end of for(r=1;r<=ROWS;r++)


fclose(WRITE_VEG);

return;

}//end OutputVegcodes

//*************************************************************************************
void OutputInitialGoal(void)
//*************************************************************************************
{
/*
After an initial Stand Goal selection has been made, this will spit out the goals in a binary
file that can be brought into ArcInfo and mapped and/or used for comparison after the heuristic
has found the final solution.

This function assumes that the initial goal assignment was inserted into the Data.Goal array by
calling InputSolution() after the random initial stuff.
*/


FILE *BinOut, *HeaderOut;
char Temp[256];

int *ptr_srp;                   //Starting Row Position
ushort   *ptr_column;
int r,c,HowMany;
int ColumnsLeft, ctr;
ushort StartColumn,OutColumn;
ushort *ptr_goal;
float *ptr_goalout;

//-------------------------- End of variable defining -------------------------------------


//Use this to store all the NODATA and actual values - so I can spit out a binary file at end of function - ready
for ArcInfo input
float (*GoalOut)[COLUMNS] = new float[ROWS][COLUMNS];
if(GoalOut == NULL)
                printf("There was NOT enough memory for GoalOut with %lu elements\n",ROWS*COLUMNS);
//Initialize
memset( GoalOut, 0, sizeof(GoalOut[0][0]) * ROWS * COLUMNS);


//============================================================================================
//        Store the Data.Goal[] data in the GoalOut[][] array and place a NODATA value in the correct
//  spots.  This is all to ease the transition into ArcInfo.  This way, I can spit out a
//  small binary file with the values and NODATA which AI can just read in.
// ============================================================================================

//Use the same procedure that is done with the other Output*() functions
for(r=1;r<=ROWS;r++)
{
        ptr_srp =                       &link[r-1][1];
        HowMany =                       *(ptr_srp+1);
        StartColumn =                   Data.GridColumn[(*ptr_srp)-1];                  //not a pointer!
        ptr_column =                    &Data.GridColumn[(*ptr_srp)-1];
        ptr_goal =                              &Data.Goal[(*ptr_srp)-1];

        //If the whole row is blank, store NODATA and goto next row
        if( *ptr_srp == FALSE )                 //means a zero was left in this spot during MakeLink
        {
                for(c=1;c<=COLUMNS;c++)
```

```
                        GoalOut[r-1][c-1] = (float)NODATA;

                continue;           //goto next row
        }


        //store NODATA for those cells before data starts
        for(c=1;c<StartColumn;c++)
                GoalOut[r-1][c-1] = (float)NODATA;

        //set some counters
        OutColumn = StartColumn;
        ctr = 0;

        //store values for area on landscape by checking
        //value in Data.GridColumn to match it with OutColumn value
        do{
                if(*ptr_column == OutColumn)
                {
                        GoalOut[r-1][OutColumn-1] = (float)*ptr_goal;

                        ptr_goal++;

                        ptr_column++;
                        OutColumn++;
                        ctr++;
                }
                else    //print out NODATA for the "gaps"
                {
                        GoalOut[r-1][OutColumn-1] = (float)NODATA;

                        OutColumn++;
                }
        }while(ctr != HowMany );

        //Check to see how many columns are left to do
        ColumnsLeft = COLUMNS - (OutColumn-1);

        if(ColumnsLeft == 0)
                continue;                   //go to next row

        //print out NODATA for those cells after the data that are left
        for(c=0;c<ColumnsLeft;c++)
                GoalOut[r-1][(OutColumn-1)+c] = (float)NODATA;

}//end of for(r=1;r<=ROWS;r++)


//=============================================================================================
//      Put a pointer at start of GoalOut and purge that data as a binary file
ptr_goalout = &GoalOut[0][0];


//Create the output Binary file and header file
sprintf(Temp, "%s%s%d\\InitGoal.bin",PREFIX,PreSimOutputDir,GOAL_TO_USE);
BinOut = fopen(Temp, "wb");

sprintf(Temp, "%s%s%d\\InitGoal.hdr",PREFIX,PreSimOutputDir,GOAL_TO_USE);
HeaderOut = fopen(Temp, "w");

//Write out the header data -- exact format for ArcInfo
fprintf(HeaderOut,"ncols\t\t%d\n",COLUMNS);
fprintf(HeaderOut,"nrows\t\t%d\n",ROWS);
fprintf(HeaderOut,"xllcorner\t%.6lf\n",F_XLL);
fprintf(HeaderOut,"yllcorner\t%.6lf\n",F_YLL);
fprintf(HeaderOut,"cellsize\t%d\n",CELLSIZE);
fprintf(HeaderOut,"NODATA_value\t%d\n",NODATA);
fprintf(HeaderOut,"byteorder\tLSBFIRST\n");


//And now write out all the records in GoalOut
fwrite(ptr_goalout, sizeof(float),ROWS*COLUMNS,BinOut);

fclose(BinOut);
fclose(HeaderOut);

//delete stuff on free store
delete [] GoalOut;

}//end OutputInitialGoal


INSECTS.CPP

/*
This souce code will hold all the functions needed to initiate and wreak havoc on
stands due to episodic insect outbreaks.

The insect disturbance is based on two components: 1st, a threshold is met (which
is a function of the weather) and 2nd, a severity is applied.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <time.h>
#include "globals.h"
#include "data.h"

//-------------------------------- EXTERNALS ------------------------------------
//defined in main.cpp
extern ulong NATLN;

//define in Misc.cpp
extern void DeleteToModify(void);

//defined in CommonDisturbance
extern void ExtractTreelist(struct TREELIST_FOR_PREMO TP[], int Count, int Per, ulong FTL);
extern void PrintNewTreelist(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                                                    int SnagCount,
ulong Treelist);
extern void UpdateDataTreelist(struct HIT_BY_DISTURB AllHit[], int AllCount);
extern void UpdateDataWithNewStandData(struct HIT_BY_DISTURB HitList[], int HitCount, struct NEW_STAND_DATA SD[],
int Unique, int Per);

//defined in StandData.cpp
extern void StandDataController(struct NEW_STAND_DATA SD[], int Count, struct  TREELIST_RECORD Records[], int
NoRecords );
extern void CalculateIndividualBasalCanopyWidth(struct TREELIST_RECORD Records[], int NoRecords);

//---------------------------------- INTERNALS-------------------------------------------------

int ApplyInsectDisturbance(int Per, int Weather, ulong FTTP);
int CountInsectHit(int Per);
int FillInsectHitList(struct HIT_BY_DISTURB HitList[], int Per);
int CountUniqueInsectHits(struct HIT_BY_DISTURB HitList[], int Count);
int FillUniqueInsectStructures(struct UNIQUE_INSECT UniqueList[], struct TREELIST_FOR_PREMO ToPremo[],
                                    struct HIT_BY_DISTURB HitList[], int Count);
void ApplyInsectSeverityCalculateStandData(struct UNIQUE_INSECT UL[], int Count, struct NEW_STAND_DATA StandData[],
int Weather);
int DougFirMortality(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                                                    int SnagCount,
ushort Pag, int Weather, struct NEW_STAND_DATA *ptr_sd);
int TrueFirMortality(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                                                    int SnagCount,
ushort Pag, int Weather, struct NEW_STAND_DATA *ptr_sd);
int PineMortality(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                                                    int SnagCount,
ushort Pag, int Weather, struct NEW_STAND_DATA *ptr_sd);
int CompareHitListForBugs(const void *ptr1, const void *ptr2);
void MapPotentialBugs(int Per);

//================================== end of function definitions for this code ================================

//*****************************************************************************
int ApplyInsectDisturbance(int Per, int Weather, ulong FTTP)
//*****************************************************************************
{
//Weather values are:
//        1 = Wet,  2 = Moderate,       3 = Mild Drought,   4 = Severe Drought

/*
      PAG values are:
-------------------------------------
1         Douglas fir          /         Dry
2         Douglas fir /        Wet
3         Jeffrey pine
4         Red fir
5         Pine / Oak
6         White fir /          Dry
7         White fir / Wet
8         Water
9         Barren

*/

int ActualPer, ArrayPer;
int a=0, HitCount, Records, Unique, Unique2;
char WeatherType[50];

//For Time information
clock_t Start, Finish;
double Duration;
//------------------------------------ End of variable defining ----------------------------------------

//Create the WeatherType string
if( Weather == 1)
          sprintf(WeatherType, "Wet");
else if(Weather == 2)
          sprintf(WeatherType, "Moderate");
else if(Weather == 3)
          sprintf(WeatherType, "Mild Drought");
else
          sprintf(WeatherType, "Severe Drought");

//REMEMBER: Per is the actual period, not array subscript - reset Per
ActualPer = Per;
ArrayPer  = Per-1;
```

```
if( Weather == 1 || Weather == 2)
{
          printf("\n\t*********************************************************************************************************
***********\n");
          printf("\t                                        INSECTS \n");
          printf("\t\t                    Not a drought period (%s)\n",WeatherType);
          printf("\t\t                   There will be NO episodic insect attacks\n");
          printf("\t*********************************************************************************************************
**********\n");

          return TRUE;
}
else
{
          printf("\n\t*********************************************************************************************************
***********\n");
          printf("\t                  Starting episodic disturbances for ... INSECTS \n");
          printf("\t\t                     This is a drought-type period (%s)\n",WeatherType);
          printf("\t\t                    There WILL be episodic insect attacks -- bzzzzz!\n");
          printf("\t*********************************************************************************************************
**********\n");
}


//Count up how many cells were hit by insects this period
HitCount = CountInsectHit(ArrayPer);
printf("\n\nFor Period %d, just counted %d cells being attacked for %.0lf
acres\n",ActualPer,HitCount,HitCount*ACREEQ);

//Print out the number of acres hit
PrintToStat(6, HitCount);

//If there are no cells getting hit by insects, then just return back to main
if( HitCount == FALSE )
{
          printf("!!! There were NO cells effected by insects - skipping INSECT DISTURBANCE routines !!!\n");
          return TRUE;
}

//create an array of structures on the free store to hold info on all the cells being hit
struct HIT_BY_DISTURB (*HitList) = new struct HIT_BY_DISTURB[HitCount];
if( HitList == NULL )
          printf("Problems allocating memory for HitList[] with %d records\n",HitCount);

//Initialize
memset( HitList, 0, sizeof(struct HIT_BY_DISTURB) * HitCount);


//Fill up the array of HitList structures
Records = FillInsectHitList(HitList, ArrayPer);
if(Records != HitCount)
          Bailout(77);


printf("\nGetting ready to sort the stands by Treelist-Goal-Hold-Pag-DougFir-TrueFir-Pine.....this will take awhile
for %lu cells\n\n",HitCount);
Start = clock();

//sort those records by:  Treelist-Goal-Hold-Pag-DougFir-TrueFir-Pine
mgsort( (void*)HitList,
          //base
              HitCount,
          //count of records
                    sizeof( struct HIT_BY_DISTURB),                                                    //size of each
record
                    0,  HitCount-1,
          //current division ( always: 0, Count-1 )
                    CompareHitListForBugs   );                                                         //compare
function

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
//printf("****Finished sorting in %.2lf seconds, now going to determine how many unique combinations of the above
sort there are\n",Duration);


//Count up how many of those records in HitList are actually unique combinations of Treelist-Goal-Hold-Pag-DougFir-
TrueFir-Pine
Unique = CountUniqueInsectHits(HitList,HitCount);
printf("!!!There were actually %d unique records that will each require a PREMO run\n",Unique);

/*
Create 3 different structures to hold various information (may share some common data, but are "packaged"
different)
Each of these 3 will hold information ONLY for those unique combinations of Treelist-Goal-Hold-Pag-DougFir-TrueFir-
Pine

1 - an array of structures to hold data pertaining to which insect type(s) are attacking and treelist values
2 - an array of structures to hold old and new treelist values to use when period is over and need to make new
Premo calls
3 - an array of structures to hold new Stand Data that will need to be updated in the Data.* arrays BEFORE next
disturbance
*/
```

```
struct UNIQUE_INSECT (*UniqueList)      = new struct UNIQUE_INSECT[Unique];
struct TREELIST_FOR_PREMO(*ToPremo) = new struct TREELIST_FOR_PREMO[Unique];
struct NEW_STAND_DATA(*StandData)       = new struct NEW_STAND_DATA[Unique];
if( UniqueList == NULL )
          printf("Problems allocating memory for UniqueList[] with %d records\n",Unique);
if( ToPremo == NULL )
          printf("Problems allocating memory for ToPremo[] with %d records\n",Unique);
if( StandData == NULL )
          printf("Problems allocating memory for StandData[] with %d records\n",Unique);
//Initialize
memset( UniqueList, 0, sizeof(struct UNIQUE_INSECT) * Unique);
memset( ToPremo, 0, sizeof(struct TREELIST_FOR_PREMO) * Unique);
memset( StandData, 0, sizeof(struct NEW_STAND_DATA) * Unique);

//Fill up the UniqueList and ToPremo structures and make sure same # of records processed
Unique2 = FillUniqueInsectStructures(UniqueList,ToPremo,HitList,HitCount);
if(Unique2 != Unique)
          Bailout(90);

//Update the treelist values in Data.Treelist[]
UpdateDataTreelist(HitList, HitCount);                                                    //REMEMBER -
HitList will be sorted by CELLID after this

//Extract the current period treelist from the appropriate prescriptions or copy from the \modified\ directory
ExtractTreelist(ToPremo,Unique,ActualPer,FTTP);

//Now apply the severity to those treelist just extracted - AND calculate new stand data for each treelist
ApplyInsectSeverityCalculateStandData(UniqueList, Unique, StandData, Weather);

//Now that StandData is filled up, send off with HitList (which must be sorted by CELLID) to modify the data in the
Data*[] arrays
UpdateDataWithNewStandData(HitList, HitCount, StandData, Unique, ArrayPer);

//Delete all the treelist files in the ToModify directory since they have been modified and now sit in \Modified\
directory
DeleteToModify();

//delete free store stuff
delete [] HitList;
delete [] UniqueList;
delete [] StandData;
delete [] ToPremo;

          return TRUE;
}//end ApplyInsectDisturbance

//*************************************************************************************************************
*************
void ApplyInsectSeverityCalculateStandData(struct UNIQUE_INSECT UL[], int Count, struct NEW_STAND_DATA StandData[],
int Weather)
//*************************************************************************************************************
*************
{
/*
       PAG values are:
-----------------------------------------
1         Douglas fir         /         Dry
2         Douglas fir /       Wet
3         Jeffrey pine
4         Red fir
5         Pine / Oak
6         White fir /         Dry
7         White fir / Wet
8         Water
9         Barren

This function will take each of the records in the array of UL[] structures, find the extracted treelist which is
sitting
in the ..\prescriptions\ToModify\* directory (with the label T_"NewTreelist".txt ).  Each treelist will be read in,
stored
in some fashion, and then specific mortality functions will come into play as a function of the PAG and which
insect
group or groups (DougFir,TrueFir,Pine) caused the treelist to get created as a unique combination in the first
place.
*/

FILE *IN;
char Temp[256];

int a, b, ReadStatus, NoRecords, NewSnagCount;
ulong Treelist;
ushort Pag;

ushort Plot, Status, Model, Report, Condition;
float Tpa, Dbh, Height, Ratio;
struct NEW_STAND_DATA *ptr_sd;

//-------------------------------- End of variable defining ----------------------------------------------

printf("\n*** Starting to apply specific mortality equations to the %d unique stands hit by insects ***\n",Count);


//Start a loop to do this for every record in the array of UL structures
for(a=0;a<Count;a++)
```

```
{
          //Set a pointer to the current StandData[] space
          ptr_sd = &StandData[a];

          //Grab the data that will identify the file needed in the ..\ToModify\* directory
          Treelist  = UL[a].NewTreelist;

          //Create a string to hold the filename - Always in the ToModDir
          sprintf(Temp, "%s%s\\T_%lu.txt",PREFIX,P_ToModDir,Treelist);

          //Open the file for reading
          IN        = fopen(Temp, "r");
          if( IN == NULL )
                    fprintf(stderr, "Opening of %s failed (ApplyInsectSeverity): %s\n",Temp, strerror(errno));


          //Go through the file and count how many lines(records) there actually are
          NoRecords=0;
          while( ReadStatus = fscanf(IN,"%hu %hu %f %hu %hu %f %f %f",&Plot,&Status,

          &Tpa,&Model,&Report,&Dbh,&Height,&Ratio) != EOF)
          {
                    NoRecords++;
                    if(Status != LIVE)            //Not a live tree so it will also have a code for the Condition
                              fscanf(IN, "%hu", &Condition);
          }//end while( ReadStatus ...)

          //Rewind back to the beginning of the file
          rewind(IN);

          //printf("There were %d lines in T_%lu.txt\n",NoRecords,Treelist);

          //Allocate free store memory for NoRecords amount of TREELIST_RECORD structures
          struct TREELIST_RECORD (*Records)       = new struct TREELIST_RECORD[NoRecords];
          if(Records == NULL)
                    printf("Problems allocating memory for Records[] with %d records\n",NoRecords);

          //Initialize
          memset( Records, 0, sizeof(struct TREELIST_RECORD) * NoRecords);

          //Also allocate memory for 100 records to hold data for NewSnags created
          struct TREELIST_RECORD(*NewSnags) = new struct TREELIST_RECORD[100];
          if(NewSnags == NULL)
                    printf("Problems allocating memory for NewSnags[] with 100 records\n");


          //Go through the current file again and fill up the array of Records
          for(b=0;b<NoRecords;b++)
          {
                    fscanf(IN,"%hu %hu %f %hu %hu %f %f %f",&Records[b].Plot, &Records[b].Status, &Records[b].Tpa,
&Records[b].Model,
                                                            &Records[b].Report, &Records[b].Dbh,
&Records[b].Height, &Records[b].Ratio);

                    if(Records[b].Status != 1)
                              fscanf(IN, "%hu",&Records[b].Condition);

          }//end for(b=0 ...)

          //Close the treelist file
          fclose(IN);

          //Send the current Records off to get individual basal area calculated - needed here to track specific
mortality for analysis
          CalculateIndividualBasalCanopyWidth(Records, NoRecords);

          //Regardless if needing DougFir, TrueFir, and/or Pine effects, get the current PAG associated with this
record
          Pag       = UL[a].Pag;

          //Reset the NewSnagCount
          NewSnagCount = 0;

          //One at a time - check to see if this file will be hit by DougFir, TrueFir, or Pine beetles - or any
combination

          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
          //                                                                    Doug fir mortality
          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
          if(UL[a].DougFir == TRUE)
          {
                    //An error checker to make sure initial breakdown of Unique combinations was correct
                    if( Pag == PAG_REDFIR || Pag == PAG_WFWET  || Pag == PAG_JEFFPINE)
                              Bailout(92);

                    NewSnagCount = DougFirMortality(Records, NoRecords, NewSnags, NewSnagCount, Pag, Weather,
ptr_sd);
          }
          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
          //                                                                    True fir mortality
          //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
          if(UL[a].TrueFir == TRUE)
          {
                    //An error checker to make sure initial breakdown of Unique combinations was correct
                    if(Pag == PAG_JEFFPINE)
```

```
                    Bailout(92);

                NewSnagCount = TrueFirMortality(Records, NoRecords, NewSnags, NewSnagCount, Pag, Weather,
ptr_sd);
        }
        //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        //                                                                          Pine mortality
        //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        if(UL[a].Pine == TRUE)
        {
                //An error checker to make sure initial breakdown of Unique combinations was correct
                if(Pag > PAG_BARREN)
                        Bailout(93);

                NewSnagCount = PineMortality(Records, NoRecords, NewSnags, NewSnagCount, Pag, Weather, ptr_sd);
        }


        //Print out the records in Records[] and NewSnags[]
        PrintNewTreelist(Records,NoRecords,NewSnags,NewSnagCount, Treelist);

        //Store the treelist value in StandData
        StandData[a].Treelist = Treelist;

        //Calculate new landscape metrics (fuel, closure, height, blc, cbd )
        StandDataController(StandData, a, Records, NoRecords);

        //delete stuff on free store
        delete [] Records;
        delete [] NewSnags;

}//end for(a=0 ...)

}//end ApplyInsectSeverity


//*************************************************************************************************
int PineMortality(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                        int SnagCount, ushort Pag, int Weather,
struct NEW_STAND_DATA *ptr_sd)
//*************************************************************************************************
{
/*
This function will apply the specific mortality effects to those treelist that are being hit
by pine insects (western pine beetle, mtn pine beetle, pine engraver).  It is assumed that the Basal Area threshold
was exceeded for this treelist, based on its Pag back in the FillHitList() function.

The NewSnags[] structures will hold data for those new snags created as a result of the mortality applied.

Weather values are:
1 = Wet,  2 = Moderate,       3 = Mild Drought,   4 = Severe Drought


       PAG values are:
-------------------------------------
1       Douglas fir         /       Dry
2       Douglas fir /       Wet
3       Jeffrey pine
4       Red fir
5       Pine / Oak
6       White fir /         Dry
7       White fir / Wet
8       Water
9       Barren

*/

int  a;
float Mort;
float MortTpa, RemainTpa,StandMortBasal=0, StandMortBigTrees=0;
struct TREELIST_RECORD *ptr_record, *ptr_snag;

//------------------------------------------- End of variable definition ------------------------------------
if(SnagCount > 99)
{
        printf("\a\a\a\a\aNeed to allocate more space for NewSnags\n");
        SnagCount = 90;                                                         //just reset and reuse
the last 10 records for now
}


//Set the mortality weight based on the incoming Weather
if(Weather == 3)    //MildDrought
        Mort = (float).1;               // 10%
else                                    //Assuming only a 4 (SevereDrought) can come in
        Mort = (float).3;

//Go through all the records in Records[] and find those that should have mortality applied
for(a=0;a<Count;a++)
{
        //Must be a live tree that is modeled as
        if(Records[a].Status == LIVE && (Records[a].Model == KPINE || Records[a].Model == PPINE ||
Records[a].Model == SPINE) )
        {
                //Set a pointer here to make it easier to copy over data into NewSnags[]
```

```
                    ptr_record = &Records[a];
                    ptr_snag   = &NewSnags[SnagCount];

                    //Calculate the MortTpa and the RemainTpa;
                    MortTpa = Mort * Records[a].Tpa;
                    RemainTpa = Records[a].Tpa - MortTpa;

                    //Calculate the BasalArea mortality
                    StandMortBasal += (MortTpa * Records[a].Basal);

                    //Track those trees >= 30" DBH and the total number killed
                    if(Records[a].Dbh >= BIG_TREE_SIZE )
                              StandMortBigTrees += MortTpa * (float)ACREEQ;
          //convert to an actual number

                    //Put the RemainTpa back into the current record
                    Records[a].Tpa = RemainTpa;

                    //copy over the current record from Records to the appropriate NewSnag record
                    memcpy(ptr_snag, ptr_record, sizeof(struct TREELIST_RECORD) );

                    //However, some values in NewSnags[].*are wrong - fill with correct values
                    NewSnags[SnagCount].Status    = SNAG;
                    NewSnags[SnagCount].Tpa       = MortTpa;
                    NewSnags[SnagCount].Condition = 1;                         //Condition code for a new snag

                    //Increment SnagCount to track the total number of snags create
                    SnagCount++;
                    if(SnagCount > 99)
                    {
                              printf("\a\a\a\a\aNeed to allocate more space for NewSnags\n");
                              SnagCount = 90;
          //just reset and reuse the last 10 records for now
                    }

          }//end if(Records[a].Status ...)
}//end for(a=0 ... )

//Cumulative track the Stand Basal Area Mortality & the Big Trees Killed
ptr_sd->BasalAreaKilled += StandMortBasal;
ptr_sd->BigTreesKilled += StandMortBigTrees;

          return SnagCount;
}//end PineMortality


//*************************************************************************************************
int TrueFirMortality(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                  int SnagCount, ushort Pag, int Weather,
struct NEW_STAND_DATA *ptr_sd)
//*************************************************************************************************
{
/*
This function will apply the specific mortality effects to those treelist that are being hit
by True fir insects (fir engraver).  It is assumed that the Basal Area threshold
was exceeded for this treelist, based on its Pag back in the FillHitList() function.

The NewSnags[] structures will hold data for those new snags created as a result of the mortality applied.

Weather values are:
1 = Wet,  2 = Moderate,       3 = Mild Drought,   4 = Severe Drought


       PAG values are:
-----------------------------------------
1          Douglas fir          /          Dry
2          Douglas fir /        Wet
3          Jeffrey pine
4          Red fir
5          Pine / Oak
6          White fir /          Dry
7          White fir / Wet
8          Water
9          Barren

*/

int  a;
float Mort;
float MortTpa, RemainTpa,StandMortBasal=0, StandMortBigTrees=0;
struct TREELIST_RECORD *ptr_record, *ptr_snag;

//--------------------------------- End of variable defining -------------------------------------------

//Set the Mort weight based on incoming Pag and the weather
if(Pag == PAG_REDFIR || Pag == PAG_WFDRY || Pag == PAG_WFWET)
{
          if(Weather == 3)                                                                        //MildDrought
                    Mort = (float).1;                                                             //10%
          else
          //Severe Drought
                    Mort = (float).2;
}
else if(Pag == PAG_DFDRY || Pag == PAG_DFWET)
{
```

```
        if(Weather == 3)                                                        //MildDrought
                Mort = (float).2;                                               //20%
        else
        //Severe Drought
                Mort = (float).4;
}
else if(Pag == PAG_PINEOAK)
{
        if(Weather == 3)                                                        //MildDrought
                Mort = (float).4;                                               //40%
        else
        //Severe Drought
                Mort = (float).6;
}
else
        Bailout(93);

//So go through all the records in Records[] and find those that will have mortality applied
for(a=0;a<Count;a++)
{
        if(Pag == PAG_REDFIR)
        {
                //Must be a live tree that is modeled as White fir or Red fir
                if(Records[a].Status == LIVE && (Records[a].Model == WFIR || Records[a].Model == RFIR) )
                {
                        //Set a pointer here to make it easier to copy over data into NewSnags[]
                        ptr_record = &Records[a];
                        ptr_snag   = &NewSnags[SnagCount];

                        //Calculate the MortTpa and the RemainTpa;
                        MortTpa = Mort * Records[a].Tpa;
                        RemainTpa = Records[a].Tpa - MortTpa;

                        //Calculate the BasalArea mortality
                        StandMortBasal += (MortTpa * Records[a].Basal);

                        //Track those trees >= 30" DBH and the total number killed
                        if(Records[a].Dbh >= BIG_TREE_SIZE )
                                StandMortBigTrees += MortTpa * (float)ACREEQ;
                //convert to an actual number

                        //Put the RemainTpa back into the current record
                        Records[a].Tpa = RemainTpa;

                        //copy over the current record from Records to the appropriate NewSnag record
                        memcpy(ptr_snag, ptr_record, sizeof(struct TREELIST_RECORD) );

                        //However, some values in NewSnags[].*are wrong - fill with correct values
                        NewSnags[SnagCount].Status    = SNAG;
                        NewSnags[SnagCount].Tpa       = MortTpa;
                        NewSnags[SnagCount].Condition = 1;                      //Condition code for a
new snag

                        //Increment SnagCount to track the total number of snags create
                        SnagCount++;
                        if(SnagCount > 99)
                        {
                                printf("\a\a\a\a\aNeed to allocate more space for NewSnags\n");
                                SnagCount = 90;
        //just reset and reuse the last 10 records for now
                        }
                }//end if(Records[a].Status ...)
        }//end if(Pag == REDFIR
        else
        {
                //Must be a live tree that is modeled as White fir
                if(Records[a].Status == LIVE && Records[a].Model == WFIR )
                {
                        //Set a pointer here to make it easier to copy over data into NewSnags[]
                        ptr_record = &Records[a];
                        ptr_snag   = &NewSnags[SnagCount];

                        //Calculate the MortTpa and the RemainTpa;
                        MortTpa = Mort * Records[a].Tpa;
                        RemainTpa = Records[a].Tpa - MortTpa;

                        //Calculate the BasalArea mortality
                        StandMortBasal += (MortTpa * Records[a].Basal);

                        //Track those trees >= 30" DBH and the total number killed
                        if(Records[a].Dbh >= BIG_TREE_SIZE )
                                StandMortBigTrees += MortTpa * (float)ACREEQ;
                //convert to an actual number

                        //Put the RemainTpa back into the current record
                        Records[a].Tpa = RemainTpa;

                        //copy over the current record from Records to the appropriate NewSnag record
                        memcpy(ptr_snag, ptr_record, sizeof(struct TREELIST_RECORD) );

                        //However, some values in NewSnags[].*are wrong - fill with correct values
                        NewSnags[SnagCount].Status    = SNAG;
                        NewSnags[SnagCount].Tpa       = MortTpa;
                        NewSnags[SnagCount].Condition = 1;                      //Condition code for a
new snag
```

```
                        //Increment SnagCount to track the total number of snags create
                        SnagCount++;
                        if(SnagCount > 99)
                        {
                                printf("\a\a\a\a\aNeed to allocate more space for NewSnags\n");
                                SnagCount = 90;
        //just reset and reuse the last 10 records for now
                        }
                }//end if(Records[a].Status ...)
        }//end if else (Pag == REDFIR

}//end for(a=0 ... )

//Cumulative track the Stand Basal Area Mortality & the Big Trees Killed
ptr_sd->BasalAreaKilled += StandMortBasal;
ptr_sd->BigTreesKilled += StandMortBigTrees;

        return SnagCount;
}//end TrueFirMortality
//*************************************************************************************************************
int DougFirMortality(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                int SnagCount, ushort Pag, int Weather, struct
NEW_STAND_DATA *ptr_sd)
//*************************************************************************************************************
{
/*
This function will apply the specific mortality effects to those treelist that are being hit
by Douglas-fir insects (DF beetle, flatheaded borer).  It is assumed that the Basal Area threshold
was exceeded for this treelist, based on its Pag back in the FillInsectHitList() function.

The NewSnags[] structures will hold data for those new snags created as a result of the mortality applied.

Weather values are:
1 = Wet,  2 = Moderate,       3 = Mild Drought,   4 = Severe Drought


       PAG values are:
-----------------------------------------
1         Douglas fir         /         Dry
2         Douglas fir /       Wet
3         Jeffrey pine
4         Red fir
5         Pine / Oak
6         White fir /         Dry
7         White fir / Wet
8         Water
9         Barren

NEW:  Keep track of the # of BigTrees > 30" that are killed and also track the amount of Basal Area killed in this
stand

*/

int  a, b=0;
float Mort;
float MortTpa, RemainTpa,StandMortBasal=0, StandMortBigTrees=0;
struct TREELIST_RECORD *ptr_record, *ptr_snag;
//------------------------------- End of variable defining ----------------------------

//Set the mortality weight based on the incoming Weather
if(Weather == 3)    //MildDrought
        Mort = (float).1;               // 10%
else                                    //Assuming only a 4 (SevereDrought) can come in
        Mort = (float).2;

//Go through all the records in Records[] and find those that should have mortality applied
for(a=0;a<Count;a++)
{
        //Must be a live tree that is modeled as Douglas-fir & have a diam > 10"
        if(Records[a].Status == LIVE && Records[a].Model == DOUGFIR && Records[a].Dbh > 10)
        {
                //Set a pointer here to make it easier to copy over data into NewSnags[]
                ptr_record = &Records[a];
                ptr_snag   = &NewSnags[SnagCount];

                //Calculate the MortTpa and the RemainTpa;
                MortTpa = Mort * Records[a].Tpa;
                RemainTpa = Records[a].Tpa - MortTpa;

                //Calculate the BasalArea mortality
                StandMortBasal += (MortTpa * Records[a].Basal);

                //Track those trees >= 30" DBH and the total number killed
                if(Records[a].Dbh >= BIG_TREE_SIZE )
                        StandMortBigTrees += MortTpa * (float)ACREEQ;
        //convert to an actual number

                //Put the RemainTpa back into the current record
                Records[a].Tpa = RemainTpa;

                //copy over the current record from Records to the appropriate NewSnag record
                memcpy(ptr_snag, ptr_record, sizeof(struct TREELIST_RECORD) );

                //However, some values in NewSnags[].*are wrong - fill with correct values
```

```
                    NewSnags[SnagCount].Status    = SNAG;
                    NewSnags[SnagCount].Tpa       = MortTpa;
                    NewSnags[SnagCount].Condition = 1;                          //Condition code for a new snag

                    //Increment SnagCount to track the total number of snags create
                    SnagCount++;
                    if(SnagCount > 99)
                    {
                              printf("\a\a\a\a\aNeed to allocate more space for NewSnags\n");
                              SnagCount = 90;
          //just reset and reuse the last 10 records for now
                    }
          }//end if(Records[a].Status ...)
}//end for(a=0 ... )

//Cumulative track the Stand Basal Area Mortality & the Big Trees Killed
ptr_sd->BasalAreaKilled += StandMortBasal;
ptr_sd->BigTreesKilled += StandMortBigTrees;

return SnagCount;

}//end DougFirMortality

//*****************************************************************************
int CountInsectHit(int Per)
//*****************************************************************************
{
//Given the current period, this function will count up how many cells will be hit according
//to guidelines provided by Jim Agee for Insect disturbances.

int a, Count;

//-------------------------- End of variable defining -----------------------------------


//Go through and count how many stands are going to be hit with insects this period
Count=0;
for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )                                      //no more cells to check
                    break;

          if(Data.Treelist[a] == NONFOREST)                                 //Not going to do anything with these because
they have no treelist anyways!
                    continue;

          //make Count by PAG and the lowest basal threshold for any of the 3 insect groups
          if(Data.Pag[a] == PAG_DFDRY || Data.Pag[a] == PAG_DFWET || Data.Pag[a] == PAG_PINEOAK)
                    Count++;
          else if(Data.Pag[a] == PAG_JEFFPINE)
          {
                    if(Data.Basal[a][Per]/BASAL_EXP > 80)
          // lowest threshold exceeded
                              Count++;
          }
          else if(Data.Pag[a] == PAG_REDFIR)
          {
                    if(Data.Basal[a][Per]/BASAL_EXP > 180)
                              Count++;
          }
          else if(Data.Pag[a] == PAG_WFDRY)
          {
                    if(Data.Basal[a][Per]/BASAL_EXP > 120)
                              Count++;
          }
          else if(Data.Pag[a] == PAG_WFWET)
          {
                    if(Data.Basal[a][Per]/BASAL_EXP > 180)
                              Count++;
          }
}//end for(a=0;a<UNIQUE;a++)

return Count;

}//end CountInsectHit

//*****************************************************************************************
int FillInsectHitList(struct HIT_BY_DISTURB HitList[], int Per)
//*****************************************************************************************
{
//Once HitList has been created in ApplyInsectDisturbance, this function will fill it up

int a, IncrementRecord, Record;
//---------------------------- end of variable defining -------------------------------------


//Now go through the entire landscape again, and this time for any cell that is being attacked by insects,
//make a "flag" of 1 in the HitList[].DougFir, HitList[].TrueFir, and/or HitList[].Pine member....also
//keep track of Treelist, Goal, Hold, Pag, and Cellid
Record=0;
for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )                                      //no more cells to check
                    break;
```

```
        if(Data.Treelist[a] == NONFOREST)                       //Not going to do anything with these because
they have no treelist anyways!
                continue;

        if(Data.Pag[a] == PAG_DFDRY )
        {
                //DFDRY gets hit no matter what (no threshold here for fir engravers - nasty!), so count it
                HitList[Record].Treelist      = Data.Treelist[a];
                HitList[Record].Goal                    = Data.Goal[a];
                HitList[Record].Hold                    = Data.Hold[a];
                HitList[Record].Pag                     = Data.Pag[a];
                HitList[Record].Cellid                  = Data.Cellid[a];

                //now flag for which of the 3 insect groups will get it
                HitList[Record].TrueFir                         = TRUE;
                //fir engravers

                if(Data.Basal[a][Per]/BASAL_EXP > 120 )
                {
                        HitList[Record].DougFir                 = TRUE;
                //DougFir beetles
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers
                }

                Record++;
        }//end DFDRY


        else if( Data.Pag[a] == PAG_DFWET)
        {
                //DFWET gets hit no matter what (no threshold here for fir engravers - nasty!), so count it
                HitList[Record].Treelist      = Data.Treelist[a];
                HitList[Record].Goal                    = Data.Goal[a];
                HitList[Record].Hold                    = Data.Hold[a];
                HitList[Record].Pag                     = Data.Pag[a];
                HitList[Record].Cellid                  = Data.Cellid[a];

                //now flag for which of the 3 insect groups will get it
                HitList[Record].TrueFir                         = TRUE;
                //fir engravers

                if(Data.Basal[a][Per]/BASAL_EXP > 250 )
                        HitList[Record].DougFir                 = TRUE;
                //DougFir beetles

                if(Data.Basal[a][Per]/BASAL_EXP > 180 )
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers

                Record++;
        }//end DFWET


        else if( Data.Pag[a] == PAG_PINEOAK)
        {
                //PINEOAK gets hit no matter what (no threshold here for fir engravers - nasty!), so count it
                HitList[Record].Treelist      = Data.Treelist[a];
                HitList[Record].Goal                    = Data.Goal[a];
                HitList[Record].Hold                    = Data.Hold[a];
                HitList[Record].Pag                     = Data.Pag[a];
                HitList[Record].Cellid                  = Data.Cellid[a];

                //now flag for which of the 3 insect groups will get it
                HitList[Record].TrueFir                         = TRUE;
                //fir engravers

                if(Data.Basal[a][Per]/BASAL_EXP > 80 )
                {
                        HitList[Record].DougFir                 = TRUE;
                //DougFir beetles
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers
                }

                Record++;
        }//end DFDRY


        else if( Data.Pag[a] == PAG_JEFFPINE)
        {
                IncrementRecord = FALSE;

                //only get mortality due to pine beetles and engraves
                if( Data.Basal[a][Per]/BASAL_EXP > 80 )
                {
                        HitList[Record].Treelist      = Data.Treelist[a];
                        HitList[Record].Goal                    = Data.Goal[a];
                        HitList[Record].Hold                    = Data.Hold[a];
                        HitList[Record].Pag                     = Data.Pag[a];
                        HitList[Record].Cellid                  = Data.Cellid[a];
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers
                        IncrementRecord                                         = TRUE;
                }
```

```
                if(IncrementRecord == TRUE)
                        Record++;

        }//end JEFFPINE


        else if( Data.Pag[a] == PAG_REDFIR)
        {
                IncrementRecord = FALSE;

                if( Data.Basal[a][Per]/BASAL_EXP > 180 )
                {
                        HitList[Record].Treelist      = Data.Treelist[a];
                        HitList[Record].Goal                    = Data.Goal[a];
                        HitList[Record].Hold                    = Data.Hold[a];
                        HitList[Record].Pag                     = Data.Pag[a];
                        HitList[Record].Cellid                  = Data.Cellid[a];
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers
                        IncrementRecord                                 = TRUE;
                }

                if( Data.Basal[a][Per]/BASAL_EXP > 250 )
                        HitList[Record].TrueFir                 = TRUE;
                //fir engravers

                if(IncrementRecord == TRUE)
                        Record++;

        }//end REDFIR


        else if( Data.Pag[a] == PAG_WFDRY)
        {
                IncrementRecord = FALSE;

                if( Data.Basal[a][Per]/BASAL_EXP > 120 )
                {
                        HitList[Record].Treelist      = Data.Treelist[a];
                        HitList[Record].Goal                    = Data.Goal[a];
                        HitList[Record].Hold                    = Data.Hold[a];
                        HitList[Record].Pag                     = Data.Pag[a];
                        HitList[Record].Cellid                  = Data.Cellid[a];
                        HitList[Record].TrueFir                 = TRUE;
                //fir engravers
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers
                        IncrementRecord                                 = TRUE;
                }

                if( Data.Basal[a][Per]/BASAL_EXP > 250 )
                        HitList[Record].DougFir                 = TRUE;
                //DougFir beetles

                if(IncrementRecord == TRUE)
                        Record++;

        }//end WFDRY


        else if( Data.Pag[a] == PAG_WFWET)
        {
                IncrementRecord = FALSE;

                if( Data.Basal[a][Per]/BASAL_EXP > 180 )
                {
                        HitList[Record].Treelist      = Data.Treelist[a];
                        HitList[Record].Goal                    = Data.Goal[a];
                        HitList[Record].Hold                    = Data.Hold[a];
                        HitList[Record].Pag                     = Data.Pag[a];
                        HitList[Record].Cellid                  = Data.Cellid[a];
                        HitList[Record].Pine                    = TRUE;
                //pine beetles and engravers
                        IncrementRecord                                 = TRUE;
                }

                if( Data.Basal[a][Per]/BASAL_EXP > 250 )
                        HitList[Record].TrueFir                 = TRUE;
                //fir engravers

                if(IncrementRecord == TRUE)
                        Record++;

        }//end WFDRY

}//end for(a=0;a<UNIQUE;a++)


return Record;

}//end FillHitList

//*****************************************************************************************
int CountUniqueInsectHits(struct HIT_BY_DISTURB HitList[], int Count)
```

```
//*****************************************************************************************
{
//Go through HitList[] and find how many actual Unique combinations of Treelist-Goal-Hold-Pag-DougFir-TrueFir-Pine
there are

int a,b,Unique;
ulong EvalTreelist;
ushort EvalGoal, EvalHold, EvalPag, EvalDougFir, EvalTrueFir, EvalPine;
//------------------------- end of variable defining ---------------------------------------

Unique = 0;
b = 0;
for(a=0;a<Count;)                                                          //a will get increment by other
loop
{
          if(b == Count)                                                                    //because of weird
incremental method, b will reach end first but a doesn't know that
                    break;

          Unique++;                                                                         //first one always counts
as do others because a gets reset in other loop

          //Set the initial Eval* variables
          EvalTreelist        = HitList[a].Treelist;
          EvalGoal            = HitList[a].Goal;
          EvalHold            = HitList[a].Hold;
          EvalPag                     = HitList[a].Pag;
          EvalDougFir                 = HitList[a].DougFir;
          EvalTrueFir                 = HitList[a].TrueFir;
          EvalPine            = HitList[a].Pine;

          //since HitList is already sorted, start at next record and look downward until no longer a match
          for(b=a+1;b<Count;)
          {
                    if(       HitList[b].Treelist == EvalTreelist &&
                              HitList[b].Goal               == EvalGoal &&
                              HitList[b].Hold               == EvalHold &&
                              HitList[b].Pag                == EvalPag &&
                              HitList[b].DougFir  == EvalDougFir &&
                              HitList[b].TrueFir  == EvalTrueFir &&
                              HitList[b].Pine               == EvalPine
                              )

                              b++;
          //look at next record
                    else
                    {
                              //Set the "a" variable to where "b" is because this is the next unique match
                              a = b;
                              break;
                    }
          }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)

return Unique;

}//end CountUniqueInsectHits

//*****************************************************************************************************************
**************************
int FillUniqueInsectStructures(struct UNIQUE_INSECT UniqueList[], struct TREELIST_FOR_PREMO ToPremo[],

                                        struct HIT_BY_DISTURB HitList[], int Count)
//*****************************************************************************************************************
**************************
{
//Go through HitList[] again and find those actual Unique combinations of Treelist-Goal-Hold-Pag-DougFir-TrueFir-
Pine counted earlier
//and this time fill up the UniqueList and ToPremo structures, as well as put the NewTreelist value in HitList[]
int a, b, Unique;
ulong EvalTreelist;
ushort EvalGoal, EvalHold, EvalPag, EvalDougFir, EvalTrueFir, EvalPine;

//---------------------------- End of variable defining ------------------------------------------

Unique = 0;
b = 0;                                                                                      //This must be
reset because above it left loop with value of Count
for(a=0;a<Count;)                                                          //a will get increment by other
loop
{
          if(b == Count)                                                                    //because of weird
incremental method, b will reach end first but a doesn't know that
                    break;

          Unique++;                                                                         //first one always counts
as do others because a gets reset in other loop

          //Set the initial Eval* variables
          EvalTreelist        = HitList[a].Treelist;
          EvalGoal            = HitList[a].Goal;
          EvalHold            = HitList[a].Hold;
          EvalPag                     = HitList[a].Pag;
          EvalDougFir                 = HitList[a].DougFir;
          EvalTrueFir                 = HitList[a].TrueFir;
```

```
        EvalPine            = HitList[a].Pine;

        //Insert those values in the array of UniqueList structures
        UniqueList[Unique-1].Treelist = EvalTreelist;
        UniqueList[Unique-1].Goal               = EvalGoal;
        UniqueList[Unique-1].Hold               = EvalHold;
        UniqueList[Unique-1].Pag                = EvalPag;
        UniqueList[Unique-1].DougFir  = EvalDougFir;
        UniqueList[Unique-1].TrueFir  = EvalTrueFir;
        UniqueList[Unique-1].Pine               = EvalPine;

        //And put the needed values in the array of ToPremo structures
        ToPremo[Unique-1].OldTreelist = EvalTreelist;
        ToPremo[Unique-1].Goal                          = EvalGoal;
        ToPremo[Unique-1].Hold                          = EvalHold;

        //Put the NATLN in for this first unique combination - this global variable is set in Main.cpp and also
used by FireEffects.cpp
        HitList[a].NewTreelist                                          = NATLN;
        UniqueList[Unique-1].NewTreelist        = NATLN;
        ToPremo[Unique-1].NewTreelist           = NATLN;

        //since HitList is already sorted, start at next record and look downward until no longer a match
        for(b=a+1;b<Count;)
        {
                if(     HitList[b].Treelist == EvalTreelist &&
                        HitList[b].Goal             == EvalGoal &&
                        HitList[b].Hold             == EvalHold &&
                        HitList[b].Pag              == EvalPag &&
                        HitList[b].DougFir  == EvalDougFir &&
                        HitList[b].TrueFir  == EvalTrueFir &&
                        HitList[b].Pine             == EvalPine
                        )
                {
                        HitList[b].NewTreelist                                                  = NATLN;
        //Also put the current NATLN in this structure
                        b++;
                                        //Then look at next record
                }
                else
                {
                        //Set the "a" variable to where "b" is because this is the next unique match
                        a = b;
                        NATLN++;
                        break;
                }
        }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)

//Always increment NATLN one more
NATLN++;

return Unique;

}//end FillUniqueInsectStructures


//*************************************************************************************************
int CompareHitListForBugs(const void *ptr1, const void *ptr2)
//*************************************************************************************************
{

        //Just to typecast them since we aren't actually passing in pointers
        struct HIT_BY_DISTURB *elem1;
        struct HIT_BY_DISTURB *elem2;

        elem1 = (struct HIT_BY_DISTURB *)ptr1;
        elem2 = (struct HIT_BY_DISTURB *)ptr2;


    if( elem1->Treelist < elem2->Treelist )
        //First sort by Treelist
        return -1;
    if( elem1->Treelist > elem2->Treelist )
        return 1;
    else
                                //Then by Goal
        {
                if( elem1->Goal < elem2->Goal )

                        return -1;
                if( elem1->Goal > elem2->Goal )
                        return 1;
                else
                                        //Then by Hold
                {
                        if( elem1->Hold < elem2->Hold )

                                return -1;
                        if( elem1->Hold > elem2->Hold )
                                return 1;
                        else
                                        //Then by Pag
                        {
```

```
                                        if( elem1->Pag < elem2->Pag )

                                                return -1;
                                        if( elem1->Pag > elem2->Pag )
                                                return 1;
                                        else
                                                //Then by DougFir
                                        {
                                                if( elem1->DougFir < elem2->DougFir )

                                                        return -1;
                                                if( elem1->DougFir > elem2->DougFir )
                                                        return 1;
                                                else
                                                //Then by TrueFir
                                                {
                                                        if( elem1->TrueFir < elem2->TrueFir )

                                                                return -1;
                                                        if( elem1->TrueFir > elem2->TrueFir )
                                                                return 1;
                                                        else
                                                //Then by Pine
                                                                {
                                                                        if( elem1->Pine < elem2->Pine )

                                                                                return -1;
                                                                        if( elem1->Pine > elem2->Pine )
                                                                                return 1;
                                                                        else
                                                                                return 0;
                                        //FINISHED!!
                                                                }//end Pine
                                                }//end TrueFir
                                        }//end DougFir
                                }//end Pag
                        }//end Hold
                }//end Goal

}//end CompareHitListForBugs

//*****************************************************************************************
void MapPotentialBugs(int Per)
//*****************************************************************************************
{
//Given the current period, this function will count up how many cells will be hit according
//to guidelines provided by Jim Agee for Insect disturbances.
/*
Using the same rules at in CountInsectHit, make a temp array (size of the landscape) that has the
value 1 for those cells that will be hit by an insect during a drought weather period.  This is
not saying anything about the severity of the effects, just whether or not it will be "hit" that period
*/

FILE *BinOut, *HeaderOut;
char Temp[256];

int ArrayPer;
int a, Count;
int Hit = 1;

int *ptr_srp;                   //Starting Row Position
ushort   *ptr_column;
int r,c,HowMany;
int ColumnsLeft, ctr;
ushort StartColumn,OutColumn;
ushort *ptr_bugs;
float *ptr_bugsout;


//-------------------------- End of variable defining ---------------------------------------

//Set the ArrayPer variable
ArrayPer = Per-1;

//Use this to store all the NODATA and actual values - so I can spit out a binary file at end of function - ready
for ArcInfo input
float (*BugsOut)[COLUMNS] = new float[ROWS][COLUMNS];
if(BugsOut == NULL)
                printf("There was NOT enough memory for BugsOut with %lu elements\n",ROWS*COLUMNS);

//Use this store just the actual values to match with the Data.*[] arrays
ushort(*BugsHit) = new ushort[UNIQUE];
if(BugsHit == NULL)
                printf("There was NOT enough memory for BugsHit with %lu elements\n",UNIQUE);

//Initialize the arrays above
memset( BugsOut, 0, sizeof(BugsOut[0][0]) * ROWS * COLUMNS);
memset( BugsHit, 0, sizeof(BugsHit[0]) * UNIQUE);

//Go through and count how many stands are going to be hit with insects this period -- track in the BugsHit[] array
Count=0;
for(a=0;a<UNIQUE;a++)
{
        if(Data.Cellid[a] == FALSE )                            //no more cells to check
                break;
```

```
        if(Data.Treelist[a] == NONFOREST)                       //Not going to do anything with these because
they have no treelist anyways!
                continue;

        //make Count by PAG and the lowest basal threshold for any of the 3 insect groups
        if(Data.Pag[a] == PAG_DFDRY || Data.Pag[a] == PAG_DFWET || Data.Pag[a] == PAG_PINEOAK)
                BugsHit[a] = Hit;
        else if(Data.Pag[a] == PAG_JEFFPINE)
        {
                if(Data.Basal[a][ArrayPer]/BASAL_EXP > 80)
                // lowest threshold exceeded
                        BugsHit[a] = Hit;
        }
        else if(Data.Pag[a] == PAG_REDFIR)
        {
                if(Data.Basal[a][ArrayPer]/BASAL_EXP > 180)
                        BugsHit[a] = Hit;
        }
        else if(Data.Pag[a] == PAG_WFDRY)
        {
                if(Data.Basal[a][ArrayPer]/BASAL_EXP > 120)
                        BugsHit[a] = Hit;
        }
        else if(Data.Pag[a] == PAG_WFWET)
        {
                if(Data.Basal[a][ArrayPer]/BASAL_EXP > 180)
                        BugsHit[a] = Hit;
        }
}//end for(a=0;a<UNIQUE;a++)


//=============================================================================================
//      Store the BugsHit data in the BugsOut[][] array and place a NODATA value in the correct
//  spots.  This is all to ease the transition into ArcInfo.  This way, I can spit out a
//  small binary file with the values and NODATA which AI can just read in.
// ============================================================================================

//Use the same procedure that is done with the other Output*() functions

for(r=1;r<=ROWS;r++)
{
        ptr_srp =                       &link[r-1][1];
        HowMany =                       *(ptr_srp+1);
        StartColumn =                   Data.GridColumn[(*ptr_srp)-1];                  //not a pointer!
        ptr_column =                    &Data.GridColumn[(*ptr_srp)-1];
        ptr_bugs =                              &BugsHit[(*ptr_srp)-1];

        //If the whole row is blank, store NODATA and goto next row
        if( *ptr_srp == FALSE )                 //means a zero was left in this spot during MakeLink
        {
                for(c=1;c<=COLUMNS;c++)
                        BugsOut[r-1][c-1] = (float)NODATA;

                continue;               //goto next row
        }


        //store NODATA for those cells before data starts
        for(c=1;c<StartColumn;c++)
                BugsOut[r-1][c-1] = (float)NODATA;

        //set some counters
        OutColumn = StartColumn;
        ctr = 0;

        //store values for area on landscape by checking
        //value in Data.GridColumn to match it with OutColumn value
        do{
                if(*ptr_column == OutColumn)
                {
                        BugsOut[r-1][OutColumn-1] = (float)*ptr_bugs;

                        ptr_bugs++;

                        ptr_column++;
                        OutColumn++;
                        ctr++;
                }
                else      //print out NODATA for the "gaps"
                {
                        BugsOut[r-1][OutColumn-1] = (float)NODATA;

                        OutColumn++;
                }
        }while(ctr != HowMany );

        //Check to see how many columns are left to do
        ColumnsLeft = COLUMNS - (OutColumn-1);

        if(ColumnsLeft == 0)
                continue;                       //go to next row

        //print out NODATA for those cells after the data that are left
        for(c=0;c<ColumnsLeft;c++)
```

```
                    BugsOut[r-1][(OutColumn-1)+c] = (float)NODATA;

}//end of for(r=1;r<=ROWS;r++)


//=============================================================================================
//        Put a pointer at start of BugsOut and purge that data as a binary file
ptr_bugsout = &BugsOut[0][0];


//Create the output Binary file and header file
sprintf(Temp, "%s%s%d\\per%d\\PotBug.bin",PREFIX,OUTPUTS,GOAL_TO_USE,Per);
BinOut = fopen(Temp, "wb");

sprintf(Temp, "%s%s%d\\per%d\\PotBug.hdr",PREFIX,OUTPUTS,GOAL_TO_USE,Per);
HeaderOut = fopen(Temp, "w");

//Write out the header data -- exact format for ArcInfo
fprintf(HeaderOut,"ncols\t\t%d\n",COLUMNS);
fprintf(HeaderOut,"nrows\t\t%d\n",ROWS);
fprintf(HeaderOut,"xllcorner\t%.6lf\n",F_XLL);
fprintf(HeaderOut,"yllcorner\t%.6lf\n",F_YLL);
fprintf(HeaderOut,"cellsize\t%d\n",CELLSIZE);
fprintf(HeaderOut,"NODATA_value\t%d\n",NODATA);
fprintf(HeaderOut,"byteorder\tLSBFIRST\n");


//And now write out all the records in BugsOut
fwrite(ptr_bugsout, sizeof(float),ROWS*COLUMNS,BinOut);

fclose(BinOut);
fclose(HeaderOut);

//delete stuff on free store
delete [] BugsHit;
delete [] BugsOut;

}//end MapPotentialBugs


ARCINFOCONTROLLER.CPP

// ************************************************************************
// This code will hold functions to do things associated with ArcInfo
// ************************************************************************

//NOTE:  22 feb 00
// These functions are being superceded by handling the AMLs individually in ArcInfo for now
// The aml's have been rewritten and could still be called up by similar functions to these
// if wanted in the future.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"


//Functions defined here in ArcResults.cpp
void VegCodeMapping(int status);
void ConvertVegcodes(void);
void VectorResults(int p);


// ************************************************************
void VegCodeMapping(int status)  // Controlling fuction //
// ************************************************************
// ************************************************************************
{                                       //if status == TRUE then do this

if(status != 0)
          ConvertVegcodes();

}//end of VegCodeMapping


// ************************************************************
void ConvertVegcodes(void)
// ************************************************************
{
          //This is to start a process to pipe a file through Arc, which in turn will run the
          //Convert_vegcodes.aml.

          //Will create two files.  1) a file called ConvertVegcodes.bat and the other called vegcodes.txt


          FILE *OpenWrite;

          char CodeFile[100];
          char BatchFile[100];
          char ToWrite[100];
          char ArcCommand[100];
          char Temp[3];
```

```
        int r;

        //Make the correct filenames and strings
        sprintf(CodeFile, "%s%s\\Vegcodes.txt",PREFIX,MapDir);
        sprintf(BatchFile, "%s%s\\ConvertVegcodes.bat",PREFIX,MapDir);
        sprintf(ToWrite, "type %s%s\\Vegcodes.txt | arc",PREFIX,MapDir);
        sprintf(ArcCommand, "&r %s%s\\convert_vegcodes.aml %s %s",PREFIX,AmlDir,MAIN_USER,ENVT);

//Find the four mapping/evaluation periods and tag them on to end of &r command
for(r=0;r<NP;r++)
{
        if(EvaluateThisPeriod[r] == TRUE)
        {
                sprintf(Temp, " %d",r+1);
                strcat(ArcCommand,Temp);
        }
}

        //Make and write the batch file
                OpenWrite = fopen(BatchFile, "w");                                              //open in write
mode
                fprintf(OpenWrite, "%s\n", ToWrite);
                fclose(OpenWrite);

        //Prepare and write the Vegcodes.txt file
                OpenWrite = fopen(CodeFile, "w");                                               //open in write
mode
                fprintf(OpenWrite, "%s\n", ArcCommand);
                fclose(OpenWrite);


        //Now run the ConvertVegcodes.bat file
        system(BatchFile);

}//end of ConvertVegcodes


//*************************************************************************************************
void VectorResults(int p)
//*************************************************************************************************
{

        //This is to create the Batch file needed to change the directory and start process to pipe
        //a file through Arc, which in turn will run the VectorResults.aml.

        //Will create two files.  1) a file called VectorResults.bat and the other called VectorResults.txt
        //which have a specific format as seen below.

        //Will then execute VectorResults.bat


        FILE *OpenWrite;
        char WriteOut[100];
        char RunBatch[100];
        char DirOut[100];
        char StartArc[150];
        char ArcCommand[100];


        //Make the filenames and command lines
        sprintf(WriteOut, "%s%s%d\\per%d\\VectorResults.bat",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(DirOut, "cd %s%s\\",PREFIX,VectorOutDir);
        sprintf(StartArc, "type %s%s%d\\per%d,VectorResults.txt | arc",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(ArcCommand, "&r %s%s\\VectorResults.aml",PREFIX,AmlDir);

                OpenWrite = fopen(WriteOut, "w");                                               //open in write
mode
                fprintf(OpenWrite, "%s%s\n", PREFIX,DirOut);
                fprintf(OpenWrite, "%s%s\n", PREFIX,StartArc);
                fclose(OpenWrite);

        //Prepare and write the VectorResults.txt file
        sprintf(WriteOut, "%s%s%d\\per%d\\VectorResults.txt",PREFIX,INPUTS,GOAL_TO_USE,p);

                OpenWrite = fopen(WriteOut, "w");                                               //open in write
mode
                fprintf(OpenWrite, "%s\n", ArcCommand);
                fclose(OpenWrite);


        //Now create and RunVectorResults to actually start the ArcInfo aml
        sprintf(RunBatch, "%s%s%d\\per%d\\VectorResults.bat",PREFIX,INPUTS,GOAL_TO_USE,p);
        system(RunBatch);

}//end VectorResults


COMMONDISTURBANCE.CPP

/*
This sourcefile will hold some functions common to any type of episodic disturbance.  Mostly to handle the
treelist and prescription data.
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"

void UpdateDataTreelist(struct HIT_BY_DISTURB AllHit[], int AllCount);
void ExtractTreelist(struct TREELIST_FOR_PREMO TP[], int Count, int Per, ulong FTL);
void PrintNewTreelist(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],int SnagCount,
ulong Treelist);
void UpdateDataWithNewStandData( struct HIT_BY_DISTURB HitList[], int HitCount, struct NEW_STAND_DATA SD[], int
Unique, int Per);
int CompareHitListCellid(const void *ptr1, const void *ptr2);
int CompareStandDataTreelist(const void *ptr1, const void *ptr2);

//================================== End of function definitions ==================================

//*****************************************************************************************************************
**************
void UpdateDataWithNewStandData( struct HIT_BY_DISTURB HitList[], int HitCount, struct NEW_STAND_DATA SD[], int
Unique, int Per)
//*****************************************************************************************************************
**************
{
/*
This function assumes that HitList was sorted by CELLID prior to coming here and that it is also filled up with the
New Treelist value.  Going one-by-one through HitList, the cellid will be checked, in ascending order, with
Data.Cellid[] values and when a match is found the New Treelist value in HitList will serve as a Key to search
in the SD[] structure for the record that has the new stand data for this treelist( which must be sorted by
"Treelist").

The incoming "Per" variable should already be in array notation

Will also update the ERA's for the cell if this is called after a fire disturbance - which can be "flagged" by
a value > 0 in  HitList[].Interval

NOTE:  This is doing NOTHING to the ERA's for those NONFOREST cells - that would have to be handled back when the
raw flame length grid is read into and stored in HitList, because nonforest stuff is discarded there!
*/
FILE *OUT;
char Temp[300];

int a,b;
ulong *ptr_cellid, *ptr_treelist;
ushort *ptr_elev, *ptr_vegcode, *ptr_basal, *ptr_closure, *ptr_cbd, *ptr_hlc, *ptr_height, *ptr_era;
double TotalBigTreesKilled=0, UnAdjustedBasalAreaKilled=0, AdjustedBasalAreaKilled;
ulong CurrentId, CurrentTreelist;

//structures
struct NEW_STAND_DATA Key;
struct NEW_STAND_DATA *ptr_record;

//-------------------------------- End of variable defining ------------------------------------

printf("*** Updating the Data.*[] arrays with new StandData after the last episodic disturbance event ***\n");
printf("** If this is for post-FIRE, then ERA's are also being adjusted to account for the associated Flame
Interval **\n");


//Initialize the key
memset( &Key, 0, sizeof( struct NEW_STAND_DATA) );


//SD[] will most likely already be sorted with lowest Treelist value first, but not a guarantee.  Sort just in
case.
qsort(    (void*)SD,                                                                    //base
          Unique,
          //count of records
                    sizeof(struct NEW_STAND_DATA),                          //size of each record
                    CompareStandDataTreelist      );                        //compare function

/*
Now that HitList and SD are both sorted by the proper member, it is possible to grab the first record in HitList[]
and look
in ASCENDING order through the Data.* arrays - because they should be in Cellid ascending order as well.
When a match is found the value in Data[].Treelist should contain the NEW treelist number (that was updated in
the earlier function UpdateDataTreelist() ).  Use this new treelist number as a Key and search through the SD[]
structures
to find a match.  When a match is found the Key can be used to copy over new stand data from SD to the Data[].*
arrays.

Because both the Data*.[] arrays and HitList[] are sorted with Cellids in ascending order, once a match has been
found
for a HitList record, there is no need to start searching from the start of Data*.[] - just increment pointer up
one.
*/


ptr_cellid                  = &Data.Cellid[0];                                  //Set pointers at first
elements in Data arrays
ptr_treelist        = &Data.Treelist[0];
ptr_elev            = &Data.Elev[0];
```

```
ptr_vegcode                     = &Data.Vegcode[0][Per];                        //Remember, "Per" already in array
notation
ptr_basal           = &Data.Basal[0][Per];
ptr_closure                     = &Data.Closure[0][Per];
ptr_cbd                         = &Data.CBDensity[0][Per];
ptr_hlc                         = &Data.HLC[0][Per];
ptr_height                      = &Data.StandHeight[0][Per];
ptr_era                         = &Data.Era[0][Per];

//Start the searching
for(a=0;a<HitCount;a++)
{
          //Grab the Cellid in HitList and also the treelist
          CurrentId             = HitList[a].Cellid;
          CurrentTreelist       = HitList[a].NewTreelist;               //Don't use the "treelist" member - that is
the old number



          //Now look for the CurrentId to match an ID in the Data.*[] arrays
          if(CurrentId != *ptr_cellid)
          {
                    do
                    {
                              ptr_cellid++;                                                         //Increment ALL
the pointers
                              ptr_treelist++;
                              ptr_elev++;

                              ptr_vegcode+=NP;                                          //Don't forget these
pointers are for 2-dimensional arrays!
                              ptr_basal+=NP;
                              ptr_closure+=NP;
                              ptr_cbd+=NP;
                              ptr_hlc+=NP;
                              ptr_height+=NP;
                              ptr_era+=NP;

                    }while(CurrentId != *ptr_cellid);
          }

          //Everything should match now - set an error checker
          if(*ptr_cellid != CurrentId || *ptr_treelist != CurrentTreelist)
                    Bailout(91);

          //Make a key using the CurrentTreelist value to search for
          Key.Treelist = CurrentTreelist;

          //Now use bsearch to find the matching record in the array of SD structures
          ptr_record = (struct NEW_STAND_DATA*)bsearch(
                    &Key,
                    (void *)SD,
                    (size_t)Unique,
                    sizeof(struct NEW_STAND_DATA),
                    CompareStandDataTreelist      );

          if(ptr_record == NULL)
                    Bailout(95);


          //*************************************************************************************************
          //******* Since we have found a match, update all the necessary values in the Data.*[] arrays *********
          //*************************************************************************************************

          //Fill in the  Data.*[] arrays and paying careful attention to typecasting - some was already done in
StandData.cpp
          //Want these to match the CONVERSION that was done in CreateSortedPremoBinaryFile()
          *ptr_basal                      = (ushort)( floor(ptr_record->Basal * BASAL_EXP ) );
          *ptr_closure            = ptr_record->Closure;
          //"floored" and converted to ushort in StandData.cpp
          *ptr_cbd                = ptr_record->Density;
          //"floored", multiplied by 100, and converted to ushort already
          *ptr_hlc                = ptr_record->HeightCrown;
          //"floored" and converted to ushort in StandData.cpp
          *ptr_height                     = ptr_record->StandHeight;
          //"floored" and converted to ushort in StandData.cpp

          //Put in the new Vegcode - be sure to check elev and modify the VegCode to be 5 or 10 if it is a MC type
          //VEGCODES are printed at end of period so always update with new codes after disturbances
          if( ptr_record->VegClass == VC_MC && ( (*ptr_elev) >= (3000*FT2M)) )
                    *ptr_vegcode = (ushort)(((ptr_record->VegClass + 5) * 100 ) + (ptr_record->Qmd * 10 ) +
(ptr_record->CoverClass));
          else
                    *ptr_vegcode = (ushort)((ptr_record->VegClass  * 100 ) + (ptr_record->Qmd * 10 ) + (ptr_record-
>CoverClass));

          //Keep track of the Total Big Trees killed and the Total Stand Basal Area
          TotalBigTreesKilled                     += (double)ptr_record->BigTreesKilled;
          UnAdjustedBasalAreaKilled     += (double)ptr_record->BasalAreaKilled;


          //NOTE:  If Jim & Bernie develop new Fuel Model class. rules for post insect or fire and they
differentiate with
          //mc>3000 and mc<3000 then this will be the place to put that in.
```

```
		//*************************************************************************************************
		//******* Check and see if ERA's need updating for after FIRE disturbances                 **********
		//*************************************************************************************************
		if(HitList[a].Interval > 0 )                                        //YES, hit by fire
		{
			/*
			Not sure what exactly to do here, but will try and mimic what John S. originally put in Premo.
That is
			the ERA for this & the next two periods will have some additional ERA added to them which will
			be a function of the Flame Length (or really the Interval since I don't keep the actual flame
length).

			NOTE:  Currently, the Data.Era[][] values are "real" values and were calculated back just
			after the Prescription selection.  They were decayed based on previous periods era first and
then new
			contributions added.  This function will not recalculate decay for periods after this
disturbance, but
			that should probably be considered.  Again, this will simply add some additional ERA
coefficient to the
			existing period values and maybe the next two periods.
			*/

			if( HitList[a].Interval > 12 )
			{
				ushort AddEra[3] = {25, 15, 5};

				for(b=0;b<3;b++)
				{
					if(Per+b == NP)
						break;

					*(ptr_era+b)        += AddEra[b];                   //DON'T increment pointer
- start of HitList loops expects it at particular spot!
				}
			}
			else if( HitList[a].Interval > 8 )
			{
				ushort AddEra[3] = {15, 5, 3};

				for(b=0;b<3;b++)
				{
					if(Per+b == NP)
						break;

					*(ptr_era+b)        += AddEra[b];                   //DON'T increment pointer
- start of HitList loops expects it at particular spot!
				}
			}
			else if( HitList[a].Interval > 4 )
				*ptr_era  += (ushort)5;
			//otherwise leave alone - no additional contribution

		}//end if(HitList ... )
}//end for(a=0... )

//Adjust the basal area mortality to get an overall average
AdjustedBasalAreaKilled = UnAdjustedBasalAreaKilled/HitCount;

//Put the mortality data in the TreeDamage.txt file opened at start of program
sprintf(Temp, "%s%s\\goal%d\\TreeDamage.txt",PREFIX,GeneralDataDir,GOAL_TO_USE);

OUT = fopen(Temp, "a+");

fprintf(OUT, "Period %d, Big Trees killed by current disturbance:     %.2lf\n",Per+1,TotalBigTreesKilled);
fprintf(OUT, "Period %d, Adjusted Avg. Basal Area killed (sq ft):     %.2lf\n",Per+1, AdjustedBasalAreaKilled);

fclose(OUT);


}//end UpdateDataWithNewStandData


//*************************************************************************************
void UpdateDataTreelist(struct HIT_BY_DISTURB AllHit[], int AllCount)
//*************************************************************************************
{
/*
This function is designed to ONLY change the treelist values in Data.Treelist.  It will NOT actually
do anything to the treelist or prescriptions themselves - that is handled by other functions.
*/
int a;
ulong CurrentId, CurrentTreelist, NewTreelistValue;
ulong *ptr_cellid, *ptr_treelist;
//ushort *ptr_flame;
//ushort FlameLength;

//For Time information
clock_t Start, Finish;
double Duration;

//-------------------------- end of variable definitions ------------------------------


//Send the AllHit structures to get sorted by their Cellid value only
Start = clock();
```

```
qsort(    (void*)AllHit,                                                                    //base
          AllCount,
          //count of records
          sizeof(struct HIT_BY_DISTURB),                                          //size of each record
          CompareHitListCellid          );                                                //compare
function

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );

//Now that the AllHit is sorted by Cellid, it is possible to grab the first record in AllHit and look
//in Ascending order through the Data.* arrays - because they should be in Cellid ascending order as well.
//When a match is found, insert the new treelist value into Data.Treelist[] and then grab the next record
//in AllHit and continue looking further down in the Data.* arrays.
ptr_cellid                = &Data.Cellid[0];
          //Set pointers at first elements of Cellid & Treelist in Data arrays
ptr_treelist        = &Data.Treelist[0];
//ptr_flame                 = &Data.Flame[0];
for(a=0;a<AllCount;a++)
{
          CurrentId                   = AllHit[a].Cellid;
          CurrentTreelist             = AllHit[a].Treelist;
          NewTreelistValue    = AllHit[a].NewTreelist;
          //FlameLength                         = AllHit[a].Interval;

          //Now look for the CurrentId to match an ID in the Data arrays
          if(CurrentId != *ptr_cellid)
          {
                    do
                    {
                              ptr_cellid++;
                              ptr_treelist++;
                              //ptr_flame++;
                    }while(CurrentId != *ptr_cellid);
          }

          //Everything should match now - set an error checker
          if(*ptr_cellid != CurrentId || *ptr_treelist != CurrentTreelist)
                    Bailout(91);

          //Since we have found a match, update the value in Data.Treelist with the NewTreelistValue
          *ptr_treelist       = NewTreelistValue;

          //*ptr_flame                = FlameLength;                          //will be zero except after a fire
type disturbance
}//end for(a=0 ...)


}//end UpdateDataTreelist

//*****************************************************************************
void ExtractTreelist(struct TREELIST_FOR_PREMO TP[], int Count, int Per, ulong FTL)
//*****************************************************************************
{
/*
The object here is to extract the current period treelist out of a prescription and send that treelist to
another directory so it can receive the particular episodic disturbance effect and later be sent through Premo.

                                                                                    - OR -

If the treelist is one that has already be hit by another disturbance this period then that treelist
was already extracted and modified and now sits in the *\model\outputs\prescriptions\modified\* directory.
All that needs to be done is to copy that treelist over to the ...*\ToModify\* directory and get yet
another new treelist number that it will be tracked by (if so, the old treelist may still be needed by another
stand so it won't be deleted - that way it can be ran through Premo and used where appropriate).
*/

int a;
ulong OldTreelist,NewTreelist;
ushort    Goal, Hold;

FILE *READ_PRESCRIPTION, *WRITE_TREELIST;
char InPrescription[256], OutTreelist[256];
char CopyFrom[256], CopyTo[256], CopyFiles[256], JunkFile[256];

double TestValue;
double Plot,Live,Tpa,Model,Report,Dbh,Height,Ratio,Dead;
int Period;
int Finished, AllRecords,GotNext;
int Status;

//---------------------------- End of variable defining ----------------------------------



//Print up a little screen information
          puts("\n\n\t\t*****************************************************************");
          puts("\t\t***                                                           ***");
          puts("\t\t***  Going to copy over treelist data so disturbance          ***");
          puts("\t\t***             effects can be made.                          ***");
          puts("\t\t***                                                           ***");
          puts("\t\t*************************************************************\n");

//Do this for each of the records found in the array of TP structures
```

```
for(a=0;a<Count;a++)
{
          OldTreelist                    = TP[a].OldTreelist;
          Goal                           = TP[a].Goal;
          Hold                           = TP[a].Hold;
          NewTreelist                    = TP[a].NewTreelist;

          //Test to see if the OldTreelist is greater than the incoming FTL variable -
          //If not, then it has not been disturbed yet this period
          if(OldTreelist < FTL )
          {

                    //If the OldTreelist is a value < NONFOREST, then the prescription data is in a different
                    //directory and so test for this and set appropriate string to use in InPrescription
                    if(OldTreelist < NONFOREST)
                              sprintf(InPrescription,
"%s%s\\P_%lu_%hu_%hu.txt",PREFIX,InitialPresDir,OldTreelist,Goal,Hold);
                    else
                              sprintf(InPrescription,
"%s%s\\P_%lu_%hu_%hu.txt",PREFIX,ModeledPresDir,OldTreelist,Goal,Hold);

                    //always send the output treelist files to the \\ToModify\\*  directory!
                    sprintf(OutTreelist, "%s%s\\T_%lu.txt",PREFIX,P_ToModDir,NewTreelist);

                    //Open up the InPrescription (which has prescription data) and find the treelist data
associated
                    //with the current period.  Then copy all those records to a new file which also needs to be
opend
                    READ_PRESCRIPTION = fopen(InPrescription, "r");
                    WRITE_TREELIST = fopen(OutTreelist,"w");
                    if (READ_PRESCRIPTION == NULL)
                              fprintf(stderr, "opening of %s failed(ExtractTreelist): %s\n",InPrescription,
strerror(errno));
                    if (WRITE_TREELIST == NULL)
                              fprintf(stderr, "opening of %s failed(ExtractTreelist): %s\n",OutTreelist,
strerror(errno));


                    //Start scanning data in and look for -9999 to indicate that a new period treelist is
                    //starting, and then verify that it is the correct period, and then scan and copy over
                    //all the treelist records from InPrescription to OutTreelist.
                    Finished = FALSE;
                    AllRecords = FALSE;

                    fscanf(READ_PRESCRIPTION, "%lf", &TestValue);               //All files must have -9999 as
first thing.

                    do{                                                                   //This do loop will
actually be broken out of by a BREAK statement

                    if(TestValue == -9999)                          //Because all files will start with this on
line 1!
                    {

                              fscanf(READ_PRESCRIPTION, "%d", &Period);

                              if(Period == Per)                     //Have the correct period - start scanning
and copying
                              {

                                        while(AllRecords == FALSE)
                                        {

                                                  if(Status = fscanf(READ_PRESCRIPTION,"%lf %lf %lf %lf %lf %lf %lf
%lf",&Plot,&Live,&Tpa,&Model,

          &Report,&Dbh,&Height,&Ratio) == EOF)
                                                            break;
                                        //Needed to stop scanning at last Period

                                                  //finished, break out this While loop
                                        if(Plot == -9999)
                    //To stop scanning in all other periods
                                                  break;
                                        //finished, break out this While loop

                                        //otherwise print to OutFile

          fprintf(WRITE_TREELIST,"%.0lf\t%.0lf\t%.2lf\t%.0lf\t%.0lf\t%.2lf\t%.2lf\t%.2lf\t",

                                                  Plot,Live,Tpa,Model,Report,Dbh,Height,Ratio);
                                        if(Live != LIVE)
                                        {
                                                  fscanf(READ_PRESCRIPTION, "%lf" , &Dead);
                    //For Snags and DWD
                                                  fprintf(WRITE_TREELIST,"%.0lf\n",Dead);
                                        }
                                        else
                                                  fprintf(WRITE_TREELIST, "\n");
                                        }//end of while(AllRecords == FALSE)


                                        break;                          //got all the records I want - quit looking
and break out of do{ loop
                              }//end of if(Period == per)
```

```
                        else
        //Scan in til the next -9999 is found
                        {
                                GotNext = FALSE;
                                while(GotNext == FALSE)
                                {
                                        fscanf(READ_PRESCRIPTION, "%lf", &TestValue);
                                        if(TestValue == -9999 || TestValue == EOF)
                                                break;
                                }
                        }//end of else

                        continue;

                }//end of if(TestValue == -9999)


                }while(Finished == FALSE);

                fclose(READ_PRESCRIPTION);
                fclose(WRITE_TREELIST);
        }//end if(OldTreelist < FTL )
        else                                                                    //The treelist has
already been hit and is in the *\modified\ directory - just copy over
        {
                //Make two strings to hold the old and new treelist path names
                sprintf(CopyFrom,"%s%s\\T_%lu.txt",PREFIX,P_ModDir,OldTreelist);
                sprintf(CopyTo,  "%s%s\\T_%lu.txt",PREFIX,P_ToModDir,NewTreelist);
                sprintf(JunkFile, "%s%s\\Junk.txt",PREFIX,P_ToModDir);

                //Make the system copy call string and execute it
                sprintf(CopyFiles, "copy %s %s > %s",CopyFrom,CopyTo,JunkFile);         //redirect
screen output to file that gets deleted later
                system(CopyFiles);
        }//end else...

}//end for(a=0 ...)

}//end ExtractTreelist



//*****************************************************************************************************************
*****************
void PrintNewTreelist(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[], int
SnagCount, ulong Treelist)
//*****************************************************************************************************************
*****************
{
/*
This function will simply print out the records in the array of Records[] and NewSnags[] structures.  They will
always be printed out to the ...\presecriptions\Modified\* directory.  From there, Premo can grab those
new treelist and do its thing - or any other disturbances during the same period can use these new treelist!
*/
/*
NOTE:  PROBLEM WITH SOME TREELIST NO HAVING LIVE TREES !!!

Having problem with episodic events (fire) wiping out all the live trees and sometimes leaving a
"Live" record with a TPA of 0.0 - and that's the only live record.  That tends to screw up things in
PREMO.  Presently I have fixed this back in ApplyFofemEffects() so go there and read notes.  However, if bugs
cause problem (they currently can't because they completly kill stands but there may be problems with rounding
of very small TPA values during Apply*BugDamage() stuff ) then something else will need to happen.
*/
FILE *OUT;
char Temp[256];
int a;
//---------------------------------- End of variable defining ----------------------------------------

//Create and open the output file
sprintf(Temp, "%s%s\\T_%lu.txt",PREFIX,P_ModDir,Treelist);
OUT = fopen(Temp, "w");
if( OUT == NULL )
        fprintf(stderr, "Opening of %s failed (PrintNewTreelist): %s\n",Temp, strerror(errno));


//Print out all the records in the array of Records
for(a=0;a<Count;a++)
{
        if(Records[a].Tpa != 0 )      //Fire will often completely kill a tree and all the data will be in
NewSnags
        {
                fprintf(OUT, "%hu\t%hu\t%.2f\t%hu\t%hu\t%.2f\t%.2f\t%.2f\t",
                                                                Records[a].Plot,
                                                                Records[a].Status,
                                                                Records[a].Tpa,
                                                                Records[a].Model,
                                                                Records[a].Report,
                                                                Records[a].Dbh,
                                                                Records[a].Height,
                                                                Records[a].Ratio
                        );

                if(Records[a].Status != LIVE)
                        fprintf(OUT, "%hu\n",Records[a].Condition);
```

```
                    else
                              fprintf(OUT, "\n");

          }//end if(Records[a].Tpa != 0 )
}//end for(a=0 ... )


//Now print out those records in the array of NewSnags
for(a=0;a<SnagCount;a++)
{
          fprintf(OUT, "%hu\t%hu\t%.2f\t%hu\t%hu\t%.2f\t%.2f\t%.2f\t",
                                                    NewSnags[a].Plot,
                                                    NewSnags[a].Status,
                                                    NewSnags[a].Tpa,
                                                    NewSnags[a].Model,
                                                    NewSnags[a].Report,
                                                    NewSnags[a].Dbh,
                                                    NewSnags[a].Height,
                                                    NewSnags[a].Ratio
                    );

          if(NewSnags[a].Status != LIVE)
                    fprintf(OUT, "%hu\n",NewSnags[a].Condition);
          else
                    fprintf(OUT, "\n");

}//end for(a=0;a<SnagCount;a++)


fclose(OUT);

}//end PrintNewTreelist

//*************************************************************************************************
int CompareHitListCellid(const void *ptr1, const void *ptr2)
//*************************************************************************************************
{
          //Just to typecast them since we aren't actually passing in pointers
          struct HIT_BY_DISTURB *elem1;
          struct HIT_BY_DISTURB *elem2;

          elem1 = (struct HIT_BY_DISTURB *)ptr1;
          elem2 = (struct HIT_BY_DISTURB *)ptr2;

    if( elem1->Cellid < elem2->Cellid )                                                    //Sort by
Cellid in ascending order
          return -1;
    if( elem1->Cellid > elem2->Cellid )
          return 1;
    else
                    return 0;
                              //Finished!!!
}//end CompareHitListCellid

//*************************************************************************************************
int CompareStandDataTreelist(const void *ptr1, const void *ptr2)
//*************************************************************************************************
{
          //Just to typecast them since we aren't actually passing in pointers
          struct NEW_STAND_DATA *elem1;
          struct NEW_STAND_DATA *elem2;

          elem1 = (struct NEW_STAND_DATA *)ptr1;
          elem2 = (struct NEW_STAND_DATA *)ptr2;

    if( elem1->Treelist < elem2->Treelist )
          //Sort by Treelist in ascending order
          return -1;
    if( elem1->Treelist > elem2->Treelist )
          return 1;
    else
                    return 0;
                              //Finished!!!
}//end CompareStandDataTreelist
```

CONSTRAINTS.CPP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"



//Defined in this Constraints.cpp file
int CheckConstraintsGoal1(struct ERA Era[], ulong NoSheds, int SubEra[]);
int CheckConstraintsGoal4(struct ERA Era[], ulong NoSheds, int SubEra[]);
```

```
//*****************************************************************************************************************
*****
int CheckConstraintsGoal1(struct ERA Era[], ulong NoSheds, int SubEra[])
//*****************************************************************************************************************
*****
{
/*
This function will check the Equivalent Roaded Acre value for all the sub-watersheds.  If any of the watersheds
violate the threshold during any period then a FALSE return value will be given.

NOTE:  remember that the Era values were inputted after being (*) by ERA_EXP, so the SubEra[] values reflects
that...
if it has a value of something like 25, it really means .25
*/

int b,x;
//---------------------------------- End of variable defining ----------------------------------


for(b=0;b<(signed)NoSheds;b++)
{
          for(x=0;x<NP;x++)
          {
                    if( Era[b].SumPeriodEra[x] / Era[b].Count > (unsigned)SubEra[x] )
                              return FALSE;
          }
}

          //If above loops complete then everything is OK
          return TRUE;
}//end CheckConstraintsGoal1


//*****************************************************************************************************************
*****
int CheckConstraintsGoal4(struct ERA Era[], ulong NoSheds, int SubEra[])
//*****************************************************************************************************************
*****
{
/*
NOTE:  This is the same as CheckConstraintsGoal1() but duplicating in case there are additional constraints for
Goal4 at a later date or if it is decided to use a different strategy.

This function will check the Equivalent Roaded Acre value for all the sub-watersheds.  If any of the watersheds
violate the threshold during any period then a FALSE return value will be given.

NOTE:  remember that the Era values were inputted after being (*) by ERA_EXP, so the SubEra[] values reflects
that...
if it has a value of something like 25, it really means .25
*/

int b,x;
//---------------------------------- End of variable defining ----------------------------------


for(b=0;b<(signed)NoSheds;b++)
{
          for(x=0;x<NP;x++)
          {
                    if( Era[b].SumPeriodEra[x] / Era[b].Count > (unsigned)SubEra[x] )
                              return FALSE;
          }
}

          //If above loops complete then everything is OK
          return TRUE;
}//end CheckConstraintsGoal4


FLAMMAPSTUFF.CPP

//*****************************************************************************************************************
//This PrepareFlammap.cpp file contains the functions that are used to prepare and run
// FLAMMAP.exe within this SafeD.exe program.
//*****************************************************************************************************************
//*****************************************************************************************************************


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"

//Functions defined here in PrepareFlammap.cpp
int PrepareFlammap(int period, int weather);
void DeleteFar(int p);
int MakeRunflammap(int p);
int PrepareLayerFile(int p);
int PrepareFlammapEnvt(int p, int w);
int WhichFlammapOutputs(int p);
void InOutFlammapResults(int p, int Status);      //Used after any Period run of Flammap
```

```
//defined in Misc.cpp
extern void CleanAndSave(int Per, int Program, int Status);

//defined in ReadData.cpp
extern long CheckHeader(int File);



//*****************************************************************************************************
int PrepareFlammap(int period, int weather)
//*****************************************************************************************************
{

//REMEMBER:  PrepareFlammap doesn't get called if #defined RERUN_SIM so is ok to delete any .far files found

          DeleteFar(period);                        //Delete any .far file created by Flammap during previous
simulation

          if( MakeRunflammap(period) )
          {
                    if( PrepareLayerFile(period) )
                    {
                              if( PrepareFlammapEnvt(period,weather) )
                              {
                                        if( WhichFlammapOutputs(period) )
                                        {
                                                  return TRUE;
          }}}}


return FALSE;

}

//*****************************************************************************************************
void DeleteFar(int p)
//*****************************************************************************************************
{
/*
Called up at three different times.
First:  Before and After any run of Flammap during RunPredictedFlammap()
Second: During a simulation period, if the simulation is not #defined RERUN_SIM then this is
                    called before a run of FLAMMAP for that period.
Third:    During a simulation period, after a run of FARSITE, if not #defined RERUN_SIM and
                    also not #defined SAVE_FOR_REUN

FARSITE and FLAMMAP will use this Layers.far file if it is there.
*/
char ToDelete[256];
FILE *t;

//------------------------------------ End of variable defining ------------------------------------

//File to check if exist
sprintf(ToDelete, "%s%s%d\\per%d\\layers.far",PREFIX,INPUTS,GOAL_TO_USE,p);


if( (t = fopen(ToDelete, "r")) != NULL)                                          //open the layers.far
file in READ mode to see if it exist
{

          fclose(t);
          sprintf(ToDelete, "del %s%s%d\\per%d\\layers.far",PREFIX,INPUTS,GOAL_TO_USE,p); //reusing the ToDelete
array!
          system(ToDelete);
}


}//end DeleteFar


//*****************************************************************************************************
int MakeRunflammap(int p)
//*****************************************************************************************************
{
          //Make the RUNFLAMMAP.BAT file needed by Flammap - on the fly

          FILE *OpenWrite;
          char WriteOut[150];

          char LayerFile[150];
          char EnvtFile[150];
          char Flammap_Outputs[150];

//------------------------------------ End of variable defining ------------------------------------

//String together the current period directory path and the appropriate file names
sprintf(LayerFile, "%s%s%d\\per%d\\layers.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
sprintf(EnvtFile, "%s%s%d\\per%d\\flammap_envt.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
sprintf(Flammap_Outputs, "%s%s%d\\per%d\\flammap_out.txt",PREFIX,OUTPUTS,GOAL_TO_USE,p);
sprintf(WriteOut, "%s%s%d\\per%d\\runflammap.bat",PREFIX,INPUTS,GOAL_TO_USE,p);

                    OpenWrite = fopen(WriteOut, "w");                                    //open in write
mode
```

```
                    fprintf(OpenWrite,"%s%s -L %s -E %s -O
%s\n",PREFIX,FlammapName,LayerFile,EnvtFile,Flammap_Outputs);
                    fclose(OpenWrite);

return TRUE;

}


//*****************************************************************************************************
int PrepareLayerFile(int p)
//*****************************************************************************************************
{
          //The layers.txt file called by Flammap specifies the files that Flammap AND Farsite will
          //use for the landscape parameters.  We are going to have to specify that the
          // HEIGHT, FUEL, BLC, DBD, and CLOSURE files are located in the current period directory
          //and the ELEV, SLOPE, and ASPECT files are in the constant directory.

          //This file DOES NOT need to exist.  It will be created from scratch using the data from below.

          //This layers.txt file will ALSO be used by Farsite during its run in this period.

          char WriteOut[50];
          FILE  *OpenWrite;

          char FuelFile[50];
          char ClosureFile [50];
          char HeightFile[50];
          char BLCFile[50];
          char CBDFile[50];

          char EUnits[10] = "meters";
          char SUnits[10] = "degrees";
          char LatFile[15] = "LATITUDE";
          char Grid[15] = "GRID_UNITS",
                                        GridUnits[10] = "meters";


//------------------------------------ End of variable defining ------------------------------------

          //Make the names of files dependent upon the current period
          sprintf(FuelFile, "%s%s%d\\per%d\\fuel.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
          sprintf(ClosureFile, "%s%s%d\\per%d\\closure.asc",PREFIX,INPUTS,GOAL_TO_USE,p);           char
          ClosureUnits[10] = "cat";
          sprintf(HeightFile, "%s%s%d\\per%d\\height.asc",PREFIX,INPUTS,GOAL_TO_USE,p);             char
          HeightUnits[10] = "feet";
          sprintf(BLCFile, "%s%s%d\\per%d\\blc.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
          char      BLCUnits[10] = "feet";
          sprintf(CBDFile, "%s%s%d\\per%d\\cbd.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
          char      CBDUnits[10] = "kg_per_m3";



          //Now create the new LAYERS.TXT file for the current period
          sprintf(WriteOut, "%s%s%d\\per%d\\layers.txt",PREFIX,INPUTS,GOAL_TO_USE,p);


          OpenWrite = fopen(WriteOut,"w");                                                          //open in write
mode
                    fprintf(OpenWrite, "%s%s\\elev_%s.asc \t\t%s\n", PREFIX,ConstantInput,ENVT,EUnits);
                    fprintf(OpenWrite, "%s%s\\slope_%s.asc \t\t%s\n", PREFIX,ConstantInput,ENVT,SUnits);
                    fprintf(OpenWrite, "%s%s\\aspect_%s.asc \t\t\n", PREFIX,ConstantInput,ENVT);
                    fprintf(OpenWrite, "%s \t\t\n", FuelFile);
                    fprintf(OpenWrite, "%s \t\t\t\t\t%s\n", ClosureFile,ClosureUnits);
                    fprintf(OpenWrite, "%s \t\t0\t\t\t%s\n", HeightFile,HeightUnits);
                    fprintf(OpenWrite, "%s \t\t0\t\t\t%s\n", BLCFile,BLCUnits);
                    fprintf(OpenWrite, "%s \t\t0\t\t\t%s\n", CBDFile,CBDUnits);
#ifdef APPLEGATE_PROJECT
                    fprintf(OpenWrite, "%s \t\t\t\t\t42\n", LatFile);
#elif defined(FRAMEWORK_PROJECT)
                    fprintf(OpenWrite, "%s \t\t\t\t\t37\n", LatFile);
#endif
                    fprintf(OpenWrite, "%s \t\t\t\t\t%s\n", Grid,GridUnits);

          fclose(OpenWrite);

return TRUE;
}

//*****************************************************************************************************
int PrepareFlammapEnvt(int p, int w)
//*****************************************************************************************************
{
          //The flammap_envt.txt file called by Flammap specifies some files that Flammap will
          //use to set up the general parameters.  We are going to need to change which FuelMoistureFile
          //uses (based on whether it is a drought year or not).
          //There may be additional reasons to change certain files at a later time.

          //This file DOES NOT need to exist.  It will be created from scratch using the data from below.
          char fms[20]="FUELMOISTURE_FILE";
          char CustomFuel[256];
          char WetFMFile[256];
          char ModFMFile[256];
          char DroFMFile[256];
```

```
          //Use these to copy whichever of the above we want to a consistent output string name
          char WriteOut[256];
          char OutMoisture[256];

          FILE  *OpenWrite;
//---------------------------------- End of variable defining -------------------------------------

          //Put together all the filenames
          sprintf(CustomFuel, "%s%s\\%s_flammap.fmd",PREFIX,ConstantInput,SHORT_NAME);
          sprintf(WetFMFile, "%s%s\\%s_wet.fms",PREFIX,ConstantInput,SHORT_NAME);
          sPrintf(ModFMFile, "%s%s\\%s_mod.fms",PREFIX,ConstantInput,SHORT_NAME);
          sprintf(DroFMFile, "%s%s\\%s_dro.fms",PREFIX,ConstantInput,SHORT_NAME);


          //Now determine which of the files are going to be used
          if(w == 1)                                                                    // is a WET
period
                    strcpy(OutMoisture, WetFMFile);
          else if(w == 2)                                                          // is a MODERATE period
                    strcpy(OutMoisture, ModFMFile);
          else
                    strcpy(OutMoisture, DroFMFile);


//21 FEB 00         - Bernie indicated we should ALWAYS run FLAMMAP with drought weather - let's reset here
          sprintf(OutMoisture, "%s",DroFMFile);


          //Create a string with the actual envt.txt file name with the full directory path
          sprintf(WriteOut, "%s%s%d\\per%d\\flammap_envt.txt",PREFIX,INPUTS,GOAL_TO_USE,p);

          OpenWrite = fopen(WriteOut, "w");                                        //open in write mode
          fprintf(OpenWrite, "FUEL_MOISTURE\t\t\t%s\n", OutMoisture);
          fprintf(OpenWrite, "CUSTOM_FUEL_MODELS\t\t%s\n",CustomFuel);
          fprintf(OpenWrite, "WIND_SPEED\t\t\t10\n");                                    //10 is a
default for now
          fprintf(OpenWrite, "WIND_DIRECTION\t\t\tUPHILL\n");                      //UPHILL is a default -
could use Azimuth degrees
          fclose(OpenWrite);

return TRUE;
}

//*************************************************************************************************
int WhichFlammapOutputs(int p)
//*************************************************************************************************
{

/*  This function will create the file "Drive":\model\outputs\per*\flammap_out.txt  which simply
has a list of files that are wanted from the Flammap program.  The possible list is that
described in the Flammap help.  The suffix (_out.txt) will be stripped off the name by Flammap and the
remaining will be used as the "basename" for what grids it generates - thus we
will always created files such as:  ...\per1\flammap.fml (for a flame length grid), etc..
*/

          FILE *OpenWrite;
          char WriteOut[50];
          char Grid1[10] = "FML";                                  //Make a Grid2, Grid3, etc., if more outputs
are wanted.

//---------------------------------- End of variable defining -------------------------------------

          //Make WriteOut dependent on the current period
          sprintf(WriteOut, "%s%s%d\\per%d\\flammap_out.txt",PREFIX,OUTPUTS,GOAL_TO_USE,p);

          OpenWrite = fopen(WriteOut, "w");                                        //open in write mode
                    fprintf(OpenWrite,"%s\n",Grid1);


          fclose(OpenWrite);

return TRUE;
}//end WhichFlammapOutputs(int p)


//*************************************************************************************************
void InOutFlammapResults(int p, int Status)
//*************************************************************************************************
{
/*
This function will open the current period run of Flammap, which produces an output flame height
file called FLAMMAP.FML in the ...\outputs\per*\ directory.  That file has values that are in meters
and this function will convert those values to the closest feet integer value.  They will then be
exported back onto the hard drive and saved, for mapping, as either p_flammap.asc or flammap.asc
(Predicted or Actual) - the original Flammap.fml file will be deleted to save space

REMEMBER - Flammap says it has a NODATA value of -1 but that is NOT true and Mark Finney is aware of the problem.
What really happens is that a 0 (zero) gets placed in those cells with NODATA, so by using Cellid as the
template I can tell which cells are really suppose to be NODATA and which are suppose to have a value of 0.

//NEW 5 Nov 99:  Delete the Flammap generated *.FML file after inputting data
*/

FILE *READ_FLAMMAP, *WriteFlammap;
char FlammapFile[250];
```

```
char garbage[100]="";
int Row,Column;
double xll, yll, junk;
int r,c,HowMany,ctr;
int *ptr_link;
char Temp[150]="";

long CellTestValue;
double FlammapTestValue;
long int FlammapNodata;
ushort FlammapConvertTest;
ushort *ptr_flammap, *ptr_gridcolumn;


FILE *BIN;
char InFile[256];
long CellidND;                                                    //hold the returned NoData value
from CheckHeader() for Cellid - it is reused

//Variable for writing the output files
int *ptr_srp;                  //Starting Row Position
ushort   *ptr_column;
int ColumnsLeft;
ushort StartColumn,OutColumn;

//---------------------------------------- End of variable defining ----------------------------------------

printf("  Preparing to import and export the FLAMMAP flame heights (import in meters, export in FEET units)\n");
puts("-----------------------------------------------------------");


//Instead of storing all Predicted and Actual flammap values in Data.* - fill this up and spit out
ushort (*FlammapValue) = new ushort [UNIQUE];
if(FlammapValue == NULL)
                    printf("There was NOT enough memory for FlammapValue with %lu elements\n",UNIQUE);


//Initialize the FlammapValue array, which will get filled with Flammap values using Cellid.bin as a guide
//to indicate those cells which were originally NODATA (because of nodata problem with Flammap).
memset( FlammapValue, 0, sizeof(FlammapValue[0]) * UNIQUE);


//Create a string to hold the name of the current input Flammap.fml file
sprintf(FlammapFile, "%s%s%d\\per%d\\flammap.fml",PREFIX,OUTPUTS,GOAL_TO_USE,p);

// ====================== OPEN AND READ THE CELLID DATA (again) ==========================================

//Create a temporary array to store the input Cellid binary data, which has data for every cell
float (*TempCellid)[COLUMNS] = new float[ROWS][COLUMNS];   //ROWS*COLUMNS is how many elements are in the initial
grid/binary file
if (TempCellid == NULL)
        printf("There was NOT enough memory for TempCellid with %lu elements\n",ROWS*COLUMNS);

//Initialize the TempCellid array
memset( TempCellid, 0, sizeof(TempCellid[0][0]) * ROWS * COLUMNS);

//Check the header data associated with this binary file and get the returned NODATA value
CellidND = CheckHeader(0);

//******read in every element of the Cellid data and store in the TempCellid array
sprintf(InFile, "%s%s\\cellid_%s.bin",PREFIX,ConstantInput,ENVT);
BIN = fopen(InFile, "rb");
if( fread(TempCellid,sizeof(TempCellid),ROWS*COLUMNS,BIN) != ROWS*COLUMNS)      //TempCellid is only a pointer!!
        Bailout(66);
else
        printf("**Binary file %s OK**\n",InFile);
fclose(BIN);



// ==================  OPEN AND READ THE FLAMMAP.FML HEADER DATA ==============================
READ_FLAMMAP = fopen(FlammapFile,"r");
if (READ_FLAMMAP == NULL)
        fprintf(stderr, "opening of %s failed:  %s\n", FlammapFile,  strerror(errno));

//use the xll and yll later on as Error Checkers
fscanf(READ_FLAMMAP,"%s %d %s %d %s %lf %s %lf %s %lf %s %ld", garbage, &Column,
        garbage, &Row, garbage, &xll,garbage,&yll,garbage,&junk,garbage,&FlammapNodata);


//Do some error checking and bail if input data is not correct
if(Column == COLUMNS && Row == ROWS)
                    printf("Rows and columns for FLAMMAP.fml are OK\n");
else
        Bailout(42);
//Do some error checking and bail if input data is not correct
if( int(xll) == XLL && int(yll) == YLL)
                    printf("X and Y origin for FLAMMAP.fml are OK\n");
else
        Bailout(43);
// ======================== End of reading header data for files
============================================================

for(r=1;r<=ROWS;r++)
{
```

```
          ptr_link = &link[r-1][1];                         //This was originally filled during ReadBinaryFiles()
          HowMany = *(ptr_link+1);

          for(c=1;c<=COLUMNS;c++)                 //c is the current column # to search for in Data.GridColumn
          {
                    //Use the TempCellid[][] as the "template" - if it has a value, then input
                    //the data found in Flammap.fml (after converting to the closest feet value)

                    //First scan everyone in - one at a time so they all are on the same cell
                    CellTestValue = (long)TempCellid[r-1][c-1];
                    fscanf(READ_FLAMMAP, "%lf",    &FlammapTestValue);


                    if(CellTestValue != CellidND)                                   //This is a VALID cell
                    {

                              //convert the flammap value
                              FlammapConvertTest = (ushort)( floor((FlammapTestValue*M2FT) + .5));

                              //Set pointer where this grid row starts in the Data.* array and in the FlammapValue
array
                              ptr_gridcolumn = &Data.GridColumn[(*ptr_link)-1];
                              ptr_flammap =        &FlammapValue[(*ptr_link)-1];


                              //look for this specific GridColumn in the Data.GridColumn array
                              for(ctr=0;ctr<HowMany;ctr++)
                              {
                                        if(*ptr_gridcolumn == (ushort)c)                  //found it
                                        {
                                                  *ptr_flammap =                FlammapConvertTest;
                                                  break;
                                        }
                                        //otherwise increment everything
                                        ptr_gridcolumn++;
                                        ptr_flammap++;
                              }//end for(ctr=0;ctr<HowMany;ctr++)

                    }//end if(CellTestValue != CellNodata)
          }//end for(c=1;c<=COLUMNS;c++)
}//end for(r=1;r<=ROWS;r++)

//close the file
fclose(READ_FLAMMAP);

//Delete the TempCellid array from free store
delete [] TempCellid;

//============================================================================
// Delete the flammap.fml file
sprintf(FlammapFile, "del %s%s%d\\per%d\\flammap.fml",PREFIX,OUTPUTS,GOAL_TO_USE,p);                //Tag on the
DELETE system command!
system(FlammapFile);

//============================================================================================================
/* Ok, the data is now stored in FlammapValue[], so just spit those values back out into an
appropriately named file (either p_*.asc for predicted values, or *.asc for actual values)
*/

//Make the correct output file name
if(Status == ACTUAL)
          sprintf(FlammapFile,"%s%s%d\\per%d\\flammap.asc",PREFIX,OUTPUTS,GOAL_TO_USE,p);
else
          sprintf(FlammapFile,"%s%s%d\\per%d\\p_flammap.asc",PREFIX,OUTPUTS,GOAL_TO_USE,p);



//open up the files to write to
WriteFlammap =                fopen(FlammapFile, "w");
if (WriteFlammap == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", FlammapFile, strerror(errno));


//write out the header data
fprintf(WriteFlammap,"ncols\t\t%d\n",COLUMNS);
fprintf(WriteFlammap,"nrows\t\t%d\n",ROWS);
fprintf(WriteFlammap,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WriteFlammap,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WriteFlammap,"cellsize\t%d\n",CELLSIZE);
fprintf(WriteFlammap,"NODATA_value\t%d\n",NODATA);

for(r=1;r<=ROWS;r++)
{
          ptr_srp =                                 &link[r-1][1];
          HowMany =                                 *(ptr_srp+1);
          StartColumn =                             Data.GridColumn[(*ptr_srp)-1];          //not a pointer!
          ptr_column =                              &Data.GridColumn[(*ptr_srp)-1];
          ptr_flammap =                             &FlammapValue[(*ptr_srp)-1];

          //If the whole row is blank, print out NODATA and goto next row
          if( *ptr_srp == FALSE )                   //means a zero was left in this spot during MakeLink
          {
                    for(c=1;c<=COLUMNS;c++)
                              fprintf(WriteFlammap,"%d  ",NODATA);
```

```
                    //put in new lines
                    fprintf(WriteFlammap, "\n");

                    continue;           //goto next row
          }


          //print out NODATA for those cells before data starts
          for(c=1;c<StartColumn;c++)
                    fprintf(WriteFlammap,"%d  ",NODATA);

          //set some counters
          OutColumn = StartColumn;
          ctr = 0;

          //print out values for area on landscape by checking
          //value in Data.GridColumn to match it with OutColumn value
          do{
                    if(*ptr_column == OutColumn)
                    {
                              fprintf(WriteFlammap, "%hu  ",*ptr_flammap);

                              ptr_flammap++;
                              ptr_column++;
                              OutColumn++;
                              ctr++;
                    }
                    else      //print out NODATA for the "gaps"
                    {
                              fprintf(WriteFlammap,"%d  ",NODATA);
                              OutColumn++;
                    }
          }while(ctr != HowMany );

          //Check to see how many columns are left to do
          ColumnsLeft = COLUMNS - (OutColumn-1);

          if(ColumnsLeft == 0)
          {
                    fprintf(WriteFlammap, "\n");
                    continue;                     //go to next row
          }

          //print out NODATA for those cells after the data that are left
          for(c=0;c<ColumnsLeft;c++)
                              fprintf(WriteFlammap,"%d  ",NODATA);

          //put in a new line
                    fprintf(WriteFlammap, "\n");

}//end of for(r=1;r<=ROWS;r++)


fclose(WriteFlammap);

//Delete the FlammapValue array on free store
delete [] FlammapValue;

}//end InOutFlammapResults


FIREEFFECTS.CPP

/* This code will apply the effects from a fire to our landscape by finding those stands that were "hit" and
modifying
their treelist thus creating new treelist for every stand that was hit.  Afterwards PREMO will need to be recalled
to
re-optimize those stand prescriptions before the landscape re-optimization takes place.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "globals.h"
#include "data.h"
//---------------------------------------- EXTERNALS -----------------------------------------
//defined in main.cpp
extern ulong NATLN;

//define in Misc.cpp
extern void DeleteToModify(void);

//defined in CommonDisturbance
extern void ExtractTreelist(struct TREELIST_FOR_PREMO TP[], int Count, int Per, ulong FTL);
extern void PrintNewTreelist(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                                                int SnagCount, ulong
Treelist);
extern void UpdateDataTreelist(struct HIT_BY_DISTURB AllHit[], int AllCount);
extern void UpdateDataWithNewStandData(struct HIT_BY_DISTURB HitList[], int HitCount, struct NEW_STAND_DATA SD[],
int Unique, int Per);

//defined in StandData.cpp
```

```
extern void StandDataController(struct NEW_STAND_DATA SD[], int Count, struct  TREELIST_RECORD Records[], int
NoRecords );
extern void CalculateIndividualBasalCanopyWidth(struct TREELIST_RECORD Records[], int NoRecords);

//defined in ReadData.cpp
extern long CheckHeader(int File);

//---------------------------------------- INTERNALS -----------------------------------------

//Declare functions used in this code
int ApplyFireDisturbance(int period,  ulong FTTP);
int CountFireHit(int per);
int FillFireHitList(struct HIT_BY_DISTURB HitList[], int Per);
int CountUniqueFireHits(struct HIT_BY_DISTURB HitList[], int Count);
int FillUniqueFireStructures(struct UNIQUE_FIRE UniqueList[], struct TREELIST_FOR_PREMO ToPremo[],

                                    struct HIT_BY_DISTURB HitList[], int Count);
void ApplyFireSeverityCalculateStandData(struct UNIQUE_FIRE UL[], int Count,  struct NEW_STAND_DATA StandData[]);
int FlameInterval(int f_feet);
void FillFofem(struct FOFEM_MATRIX *Fofem);
int ApplyFofemEffects(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                                        int SnagCount, struct FOFEM_MATRIX *Fofem, int
Interval, struct NEW_STAND_DATA *ptr_sd);
int CompareHitListForFlame(const void *ptr1, const void *ptr2);
void DoubleCheckVegcodes(void);
// ********************************************************************************************************************
// ********************************************************************************************************************

                                                // Controlling fuction //
// ************************************************************
int ApplyFireDisturbance(int period,  ulong FTTP)
// ************************************************************
{
int a=0;
int ActualPer, ArrayPer;
int Count, Records, Unique, Unique2;

//For Time information
clock_t Start, Finish;
double Duration;
//---------------------------------- End of variable defining --------------------------------------

ActualPer = period;
ArrayPer  = period - 1;

//Count up how many cells were hit by fire this period
Count = CountFireHit(ActualPer);
printf("\n\nFor Period %d, just counted %d cells that were hit by fire, for %.0lf
acres\n",ActualPer,Count,Count*ACREEQ);

//Print out the number of acres hit
PrintToStat(5, Count);

//If there are no cells getting hit by fire, then just return back to main
if( Count == FALSE )
{
          printf("!!! There were NO cells hit by fire - skipping FIRE DISTURBANCE routines !!!\n");
          return TRUE;
}


//create an array of structures on the free store to hold info on all the cells being hit
struct HIT_BY_DISTURB (*HitList) = new struct HIT_BY_DISTURB[Count];
if( HitList == NULL )
          printf("Problems allocating memory for HitList[] with %d records\n",Count);

//Initialize
memset( HitList, 0, sizeof(struct HIT_BY_DISTURB) * Count);


//Fill up the array of HitList structures
Records = FillFireHitList(HitList, ActualPer);
if(Records != Count)
          Bailout(77);


//sort those records by:  Treelist-Goal-Hold-Interval
printf("\nGetting ready to sort the stands by Treelist-Goal-Hold-Interval.....this will take awhile for %lu
cells\n\n",Count);
Start = clock();

mgsort( (void*)HitList,
          //base
                    Count,
                    //count of records
                    sizeof( struct HIT_BY_DISTURB),                                          //size of each
record
                    0,  Count-1,
          //current division ( always: 0, Count-1 )
                    CompareHitListForFlame   );                                              //compare
function

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
```

```
//Count up how many of those records in HitList are actually unique combinations of Treelist-Goal-Hold-Interval
Unique = CountUniqueFireHits(HitList,Count);
printf("!!!There were actually %d unique records that were hit by FIRE this period run\n",Unique);

/*
Create 3 different structures to hold various information (may share some common data, but are "packaged"
different)
Each of these 3 will hold information ONLY for those unique combinations of Treelist-Goal-Hold-Interval

1 - an array of structures to hold data pertaining to which fire interval and T-G-H combination, and treelist
values
2 - an array of structures to hold old and new treelist values to use when period is over and need to make new
Premo calls
3 - an array of structures to hold new Stand Data that will need to be updated in the Data.* arrays BEFORE next
disturbance
*/
struct UNIQUE_FIRE (*UniqueList) = new struct UNIQUE_FIRE[Unique];
struct TREELIST_FOR_PREMO(*ToPremo) = new struct TREELIST_FOR_PREMO[Unique];
struct NEW_STAND_DATA(*StandData)       = new struct NEW_STAND_DATA[Unique];
if( UniqueList == NULL )
          printf("Problems allocating memory for UniqueList[] with %d records\n",Unique);
if( ToPremo == NULL )
          printf("Problems allocating memory for ToPremo[] with %d records\n",Unique);
if( StandData == NULL )
          printf("Problems allocating memory for StandData[] with %d records\n",Unique);
//Initialize
memset( UniqueList, 0, sizeof(struct UNIQUE_FIRE) * Unique);
memset( ToPremo, 0, sizeof(struct TREELIST_FOR_PREMO) * Unique);
memset( StandData, 0, sizeof(struct NEW_STAND_DATA) * Unique);


//Fill up the UniqueList and ToPremo structures and make sure same # of records processed
Unique2 = FillUniqueFireStructures(UniqueList,ToPremo,HitList,Count);
if(Unique2 != Unique)
          Bailout(90);


//Update the treelist values in Data.Treelist[]
UpdateDataTreelist(HitList, Count);                                                    //REMEMBER - HitList will
be sorted by CELLID after this

//Extract the current period treelist from the appropriate prescriptions or copy from the \modified\ directory
ExtractTreelist(ToPremo,Unique,ActualPer,FTTP);

//Now apply the severity to those treelist just extracted
ApplyFireSeverityCalculateStandData(UniqueList, Unique, StandData);

//Now that StandData is filled up, send off with HitList (which must be sorted by CELLID) to modify the data in the
Data*[] arrays
UpdateDataWithNewStandData(HitList, Count, StandData, Unique, ArrayPer);

//Delete all the treelist files in the ToModify directory since they have been modified and now sit in \Modified\
directory
DeleteToModify();

//DoubleCheckVegcodes();

//Delete stuff on free store
delete [] HitList;
delete [] UniqueList;
delete [] ToPremo;
delete [] StandData;

          return TRUE;
}//end ApplyFireDisturbance

//*****************************************************************************************************************
void ApplyFireSeverityCalculateStandData(struct UNIQUE_FIRE UL[], int Count,  struct NEW_STAND_DATA StandData[])
//*****************************************************************************************************************
{
/*
This function will take each of the records in the array of UL[] structures, find the extracted
treelist which is sitting in the ..\prescriptions\ToModify\* directory (with the label
T_"NewTreelist".txt ).  Each treelist will be read in, stored in some fashion,
and then specific FOFEM mortality functions will come into play as a function of the Flame Length Interval
which caused the treelist to get created as a unique combination in the first place.
*/

FILE *IN;
char Temp[256];

int a, b, ReadStatus, NoRecords, NewSnagCount;
ulong Treelist;
ushort Interval;

ushort Plot, Status, Model, Report, Condition;
float Tpa, Dbh, Height, Ratio;
struct NEW_STAND_DATA *ptr_sd;

//------------------------------ End of variable defining ----------------------------------------------

printf("\n*** Starting to apply specific FOFEM effects to the %d unique stands hit by fire ***\n",Count);
```

```
//First thing, allocate memory for the FOFEM coefficients - sorta redundant to do every period but is quick
struct FOFEM_MATRIX (*Fofem)  = new struct FOFEM_MATRIX;
if(Fofem == NULL)
          printf("Problems allocating memory for a FOFEM_MATIX structure!\n");

//Initialize the Fofem structure
memset(Fofem, 0, sizeof(struct FOFEM_MATRIX) );

//Fill the Fofem structure up with the correct coefficients
FillFofem(Fofem);


//Start a loop to do this for every record in the array of UL structures
for(a=0;a<Count;a++)
{
          //Set a pointer to the current StandData[] space
          ptr_sd = &StandData[a];

          //Grab the data that will identify the file needed in the ..\ToModify\* directory
          Treelist  = UL[a].NewTreelist;

          //Create a string to hold the filename - Always in the ToModDir
          sprintf(Temp, "%s%s\\T_%lu.txt",PREFIX,P_ToModDir,Treelist);

          //Open the file for reading
          IN        = fopen(Temp, "r");
          if( IN == NULL )
                    fprintf(stderr, "Opening of %s failed (ApplyFireSeverity): %s\n",Temp, strerror(errno));


          //Go through the file and count how many lines(records) there actually are
          NoRecords=0;
          while( ReadStatus = fscanf(IN,"%hu %hu %f %hu %hu %f %f %f",&Plot,&Status,

          &Tpa,&Model,&Report,&Dbh,&Height,&Ratio) != EOF)
          {
                    NoRecords++;
                    if(Status != LIVE)            //Not a live tree so it will also have a code for the Condition
                              fscanf(IN, "%hu", &Condition);
          }//end while( ReadStatus ...)

          //Rewind back to the beginning of the file
          rewind(IN);

          //printf("There were %d lines in T_%lu.txt\n",NoRecords,Treelist);

          //Allocate free store memory for NoRecords amount of TREELIST_RECORD structures
          struct TREELIST_RECORD (*Records)       = new struct TREELIST_RECORD[NoRecords];
          if(Records == NULL)
                    printf("Problems allocating memory for Records[] with %d records\n",NoRecords);

          //Initialize
          memset( Records, 0, sizeof(struct TREELIST_RECORD) * NoRecords);

          //Also allocate memory to hold data for NewSnags created (a fire may create snags for every record except
those that are already Snags and DWD)
          struct TREELIST_RECORD(*NewSnags) = new struct TREELIST_RECORD[NoRecords];
          if(NewSnags == NULL)
                    printf("Problems allocating memory for NewSnags[] with %d records\n", NoRecords);


          //Go through the current file again and fill up the array of Records
          for(b=0;b<NoRecords;b++)
          {
                    fscanf(IN,"%hu %hu %f %hu %hu %f %f %f",&Records[b].Plot, &Records[b].Status, &Records[b].Tpa,
&Records[b].Model,
                                                            &Records[b].Report, &Records[b].Dbh,
&Records[b].Height, &Records[b].Ratio);

                    if(Records[b].Status != 1)
                              fscanf(IN, "%hu",&Records[b].Condition);

          }//end for(b=0 ...)

          //Close the treelist file
          fclose(IN);

          //Send the current Records off to get individual basal area calculated - needed here to track specific
mortality for analysis
          CalculateIndividualBasalCanopyWidth(Records, NoRecords);

          //Get the current Interval associated with this record
          Interval  = UL[a].Interval;

          //Reset the NewSnagCount
          NewSnagCount = 0;

          //Send the data off to have FOFEM effects applied
          NewSnagCount = ApplyFofemEffects(Records, NoRecords, NewSnags, NewSnagCount, Fofem, Interval, ptr_sd);

          //Print out the records in Records[] and NewSnags[]
          PrintNewTreelist(Records,NoRecords,NewSnags,NewSnagCount, Treelist);

          //Store the treelist value in StandData
          StandData[a].Treelist = Treelist;
```

```
          //Calculate new landscape metrics (fuel, closure, height, blc, cbd )
          StandDataController(StandData, a, Records, NoRecords);


          //delete stuff on free store
          delete [] Records;
          delete [] NewSnags;

}//end for(a=0 ...)

//Lastly, delete the Fofem structure
delete [] Fofem;

}//End ApplyFireSeverity


//*************************************************************************************************
int ApplyFofemEffects(struct TREELIST_RECORD Records[], int Count, struct TREELIST_RECORD NewSnags[],
                                    int SnagCount, struct FOFEM_MATRIX *Fofem, int Interval, struct
NEW_STAND_DATA *ptr_sd)
//*************************************************************************************************
{
/*
Look at all the individual records currently in the array of Records structures.  Depending on the
flame length interval that was passed in, apply a particular FOFEM coefficient to that record.

For those newly created snags, put that information in the array of NewSnags structures
*/
int a, DbhRow, FlameColumn, SaveSpot;
float MortTpa, RemainTpa,StandMortBasal=0, StandMortBigTrees=0;
double *ptr_fofem;
struct TREELIST_RECORD SaveRecord;
struct TREELIST_RECORD *ptr_record, *ptr_snag, *ptr_saverecord;
int AlreadySavedOne=FALSE, HadSevereMortality=FALSE;
int Fix;
//-------------------------------- End of variable defining -----------------------------------

//Start a loop to look at each record in the array of Records structures
for(a=0;a<Count;a++)
{
          //Must be a live tree
          if(Records[a].Status == LIVE )
          {

                    //Figure out which DBH row in the Fofem structure arrays to use
                    //The arrays have rows for DBH's:  1,2,4,6,8,10.....40 (array subscript 0-20)
                    if(Records[a].Dbh < 2)
                              DbhRow = 0;
                    else
                              DbhRow = (int)floor(Records[a].Dbh / 2);

                    if(DbhRow > 20)                         //Just use values for those with DBH of 40
                              DbhRow = 20;

                    //Figure out which Interval column in the Fofem structure arrays to use
                    //The Interval variable should already be a multiple of 2 - generated in FlameInterval()
function
                    //The arrays in the Fofem structure have columns for Intervals: 2,4,6,8...16 (array subscript
0-7)
                    FlameColumn = (Interval / 2) - 1;
                    if(FlameColumn > 7)           //this is a flame of over 16' - just use the 16' effects
                              FlameColumn = 7;


                    //Put a pointer at the appropriate FOFEM array to get the Mortality Coefficient
                    //associated for the Species, given its DBH, and the current Flame Interval.
                    //Check to make sure the Model code is valid
                    if(Records[a].Model > 9)
                              Bailout(33);


                    switch(Records[a].Model)      //This is the "model code" reported by Premo
                    {
                    case 0: ptr_fofem = &Fofem->BO[DbhRow][FlameColumn];  break;                    //Use the BO
array
                    case 1: ptr_fofem = &Fofem->DF[DbhRow][FlameColumn];  break;                    //Use the DF
array
                    case 2: ptr_fofem = &Fofem->DF[DbhRow][FlameColumn];  break;                    //Use the DF
array
                    case 3: ptr_fofem = &Fofem->PP[DbhRow][FlameColumn];  break;                    //Use the PP
array
                    case 4: ptr_fofem = &Fofem->HW[DbhRow][FlameColumn];  break;                    //Use the HW
array
                    case 5: ptr_fofem = &Fofem->PP[DbhRow][FlameColumn];  break;                    //Use the PP
array
                    case 6: ptr_fofem = &Fofem->WF[DbhRow][FlameColumn];  break;                    //Use the WF
array
                    case 7: ptr_fofem = &Fofem->SP[DbhRow][FlameColumn];  break;                    //Use the SP
array
                    case 8: ptr_fofem = &Fofem->HW[DbhRow][FlameColumn];  break;                    //Use the HW
array
                    case 9: ptr_fofem = &Fofem->WF[DbhRow][FlameColumn];  break;                    //Use the WF
array
                    }
```

```
                    if( *ptr_fofem != 0)                    //When 0, there is no FOFEM effect so skip this
                    {
                              /*
                              Determine the FOFEM mortality for a record - that is, if ptr_fofem is .9, then 90% of
the TPA
                              associated with the current record will die and turn into snags.  These trees will
inherit
                              the same attributes as before death(dbh,height,crown) and will get a new code to
indicate
                              what the "condition" is and a new "Status" value.  The remaining trees not affected
by the
                              mortality percentage (i.e. 10% of the TPA from the above example) will be outputted
with
                              a new TPA and the same attributes (dbh,height,crown) as before.
                              */
                              //Set a pointer here to make it easier to copy over data into NewSnags[]
                              ptr_record = &Records[a];
                              ptr_snag   = &NewSnags[SnagCount];

                              //Calculate the MortTpa and the RemainTpa;
                              MortTpa = (float)((*ptr_fofem) * Records[a].Tpa);
                              RemainTpa = Records[a].Tpa - MortTpa;

                              //Calculate the BasalArea mortality
                              StandMortBasal += (MortTpa * Records[a].Basal);

                              //Track those trees >= 30" DBH and the total number killed
                              if(Records[a].Dbh >= BIG_TREE_SIZE )
                                        StandMortBigTrees += MortTpa * (float)ACREEQ;
                    //convert to an actual number

                              //Put the RemainTpa back into the current record
                              Records[a].Tpa = RemainTpa;

                              //copy over the current record from Records to the appropriate NewSnag record
                              memcpy(ptr_snag, ptr_record, sizeof(struct TREELIST_RECORD) );

                              //If the mortality was 100% just zero out the whole record - PrintNewTreelist() won't
print those with TPA == 0.0
                              if( *ptr_fofem == 1)
                              {
                                        HadSevereMortality = TRUE;

                                        // ********************** PART I of no live trees fix
************************************************
                                        //Save the first record that gets completely wiped out - may need to
reinsert if no LIVE trees at end
                                        if(AlreadySavedOne == FALSE)
                                        {
                                                  //Place pointer at the SaveRecord structure
                                                  ptr_saverecord = &SaveRecord;
                                                  //Copy current data from Records to SaveRecord
                                                  memcpy(ptr_saverecord, ptr_record, sizeof(struct TREELIST_RECORD)
);
                                                  //Remember where this record is in the array of structures
                                                  SaveSpot = a;
                                                  //Put the original TPA back in - remember, only doing this for
one record so although BOGUS, it's livable as a fix
                                                  ptr_saverecord->Tpa = MortTpa+1;        //just because these are
usually very small stands and I think rounding problems
                                                  AlreadySavedOne = TRUE;
                                        }
                                        // ************************* end Part I fix
************************************************************

                                        //Always reset the current record to zero if complete mortality from FOFEM
                                        memset(ptr_record, 0, sizeof(struct TREELIST_RECORD) );
                              }

                              //However, some values in NewSnags[].*are wrong - fill with correct values
                              NewSnags[SnagCount].Status    = SNAG;
                              NewSnags[SnagCount].Tpa       = MortTpa;
                              NewSnags[SnagCount].Condition = 1;                        //Condition code for a
new snag - may want to change since it was a fire?

                              //Increment SnagCount to track the total number of snags create
                              SnagCount++;
                    }

          }//end if(Records[a].Status == LIVE)

}//end for(a=0 ...)

//Cumulative track the Stand Basal Area Mortality & the Big Trees Killed
ptr_sd->BasalAreaKilled += StandMortBasal;
ptr_sd->BigTreesKilled += StandMortBigTrees;

//********************************** PART II of NO LIVE TREES FIX ********************************************
/*
A problem has occurred when there is complete mortality to some records in a treelist and sometimes no
"live" trees are left in the treelist - they all got sent to snags.  Check two things:
1 - was there SevereMortality.  If so, look at all the records in Records and see
2 - is there at least ONE valid live tree with a valid TPA value that won't screw PREMO up.
```

```
If not, then simply reinsert the SaveRecord values back into the Records[] and hope that takes care of it.
*/

if( HadSevereMortality == TRUE)
{
          Fix = TRUE;

          for(a=0;a<Count;a++)
          {

                    //If there is at least one of these then no need to do any fixing
                    if(Records[a].Status == LIVE )                    //Must be a live tree
                    {
                              if(Records[a].Tpa > 0 )                              //and have a valid tpa
                              {
                                        Fix = FALSE;
                                        break;
                              }
                    }
          }//end for(a=0;a<Count;a++)

          if( Fix == TRUE )
          {
                    ptr_record = &Records[SaveSpot];
                    memcpy(ptr_record, ptr_saverecord, sizeof(struct TREELIST_RECORD));
          }

}// end if( HadSevereMortality == TRUE)
// ************************************** End Part II of fix for no Live Trees
****************************************


return SnagCount;

}//end ApplyFofemEffects


//*********************************************************************************************
void FillFofem(struct FOFEM_MATRIX *Fofem)
//*********************************************************************************************
{
/*
This function is called once every period to fill up the Fofem structure.  That structure will
contain the FOFEM coefficients developed by Jim Agee and Bernie Bahro.  Currently there are
6 different "categories" of coefficient matricies:  Black Oak, Douglas fir, Hardwoods,
Ponderosa Pine, Sugar Pine, and White fir.  These categories will have to be used for all our
stands that are hit.
*/

FILE *READ_FOFEM;
char Temp[256];

int a,b;
double *ptr_fofem;

//--------------------------- End of variable defining ---------------------------------------


//First open up the fofem.txt file
sprintf(Temp, "%s%s\\FOFEM.txt",PREFIX,ConstantInput);
READ_FOFEM =  fopen(Temp, "r");
if (READ_FOFEM == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", Temp, strerror(errno));
else
{
          printf("\n          ***************************************\n");
          printf("          **** Reading in the FOFEM.txt file ****\n");
          printf("          ***************************************\n\n");
}



for(a=0; a<6; a++)
{
                    switch(a)
                    {
                    case 0: ptr_fofem = &Fofem->BO[0][0];  break;               //Fill up BO array
                    case 1: ptr_fofem = &Fofem->DF[0][0];  break;               //Fill up DF array
                    case 2: ptr_fofem = &Fofem->HW[0][0];  break;               //Fill up HW array
                    case 3: ptr_fofem = &Fofem->PP[0][0];  break;               //Fill up PP array
                    case 4: ptr_fofem = &Fofem->SP[0][0];  break;               //Fill up SP array
                    case 5: ptr_fofem = &Fofem->WF[0][0];  break;               //Fill up WF array
                    }

                    for(b=0;b<(21*8);b++)
                    {
                              fscanf(READ_FOFEM, "%lf", ptr_fofem);
                              ptr_fofem++;
                    }

}//end of for(a=0; a<6; a++)


//Test Print out
/*
```

```
int z;

for(a=0; a<6; a++)
{
                    switch(a)
                    {
                    case 0: ptr_fofem = &Fofem->BO[0][0];  printf("\n The BO array\n"); break;
          //Read BO array
                    case 1: ptr_fofem = &Fofem->DF[0][0];  printf("\n The DF array\n"); break;
          //Read DF array
                    case 2: ptr_fofem = &Fofem->HW[0][0];  printf("\n The HW array\n"); break;
          //Read HW array
                    case 3: ptr_fofem = &Fofem->PP[0][0];  printf("\n The PP array\n"); break;
          //Read PP array
                    case 4: ptr_fofem = &Fofem->SP[0][0];  printf("\n The SP array\n"); break;
          //Read SP array
                    case 5: ptr_fofem = &Fofem->WF[0][0];  printf("\n The WF array\n"); break;
          //Read WF array
                    }

                    for(b=0;b<21;b++)
                    {
                              for(z=0;z<8;z++)
                              {
                                        printf("%.2lf  ",*ptr_fofem);
                                        ptr_fofem++;
                              }
                              printf("\n");
                    }
}//end of for(a=0; a<6; a++)
*/

fclose(READ_FOFEM);

}//end FillFofem


//*************************************************************************************************************************
*****************************
int FillUniqueFireStructures(struct UNIQUE_FIRE UniqueList[], struct TREELIST_FOR_PREMO ToPremo[], struct
HIT_BY_DISTURB HitList[],


                    int Count)
//*************************************************************************************************************************
*****************************
{
//Go through HitList[] again and find those actual Unique combinations of Treelist-Goal-Hold-Interval counted
earlier
//and this time fill up the UniqueList and ToPremo structures, as well as put the NewTreelist value in HitList[]
int a, b, Unique;
ulong EvalTreelist;
ushort EvalGoal, EvalHold, EvalInterval;

//---------------------------- End of variable defining ----------------------------------------------
Unique = 0;
b = 0;                                                                                          //This must be
reset because above it left loop with value of Count
for(a=0;a<Count;)                                                                  //a will get increment by other
loop
{
          if(b == Count)                                                                        //because of weird
incremental method, b will reach end first but a doesn't know that
                    break;

          Unique++;                                                                             //first one always counts
as do others because a gets reset in other loop

          //Set the initial Eval* variables
          EvalTreelist        = HitList[a].Treelist;
          EvalGoal            = HitList[a].Goal;
          EvalHold            = HitList[a].Hold;
          EvalInterval        = HitList[a].Interval;


          //Insert those values in the array of UniqueList structures
          UniqueList[Unique-1].Treelist = EvalTreelist;
          UniqueList[Unique-1].Goal               = EvalGoal;
          UniqueList[Unique-1].Hold               = EvalHold;
          UniqueList[Unique-1].Interval = EvalInterval;


          //And put the needed values in the array of ToPremo structures
          ToPremo[Unique-1].OldTreelist = EvalTreelist;
          ToPremo[Unique-1].Goal                          = EvalGoal;
          ToPremo[Unique-1].Hold                          = EvalHold;


          //Put the NATLN in for this first unique combination - this global variable is set in Main.cpp and also
used by Insects.cpp
          HitList[a].NewTreelist                                        = NATLN;
          UniqueList[Unique-1].NewTreelist        = NATLN;
          ToPremo[Unique-1].NewTreelist           = NATLN;
```

```
        //sine HitList is already sorted, start at next record and look downward until no longer a match
        for(b=a+1;b<Count;)
        {
                if(     HitList[b].Treelist == EvalTreelist &&
                        HitList[b].Goal             == EvalGoal &&
                        HitList[b].Hold             == EvalHold &&
                        HitList[b].Interval == EvalInterval     )
                {
                        HitList[b].NewTreelist                                          = NATLN;
        //Also put the current NATLN in this structure
                        b++;
                                        //Then look at next record
                }
                else
                {
                        //Set the "a" variable to where "b" is because this is the next unique match
                        a = b;
                        NATLN++;
                        break;
                }
        }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)


//Always increment NATLN one more
NATLN++;

return Unique;
}//end FillUniqueFireStructures


//*************************************************************************************
int CountUniqueFireHits(struct HIT_BY_DISTURB HitList[], int Count)
//*************************************************************************************
{
//Go through HitList[] and find how many actual Unique combinations of Treelist-Goal-Hold-Interval

int a,b,Unique;
ulong EvalTreelist;
ushort EvalGoal, EvalHold, EvalInterval;
//------------------------ end of variable defining -------------------------------------

Unique = 0;
b = 0;
for(a=0;a<Count;)                                                       //a will get increment by other
loop
{
        if(b == Count)                                                                  //because of weird
incremental method, b will reach end first but a doesn't know that
                break;

        Unique++;                                                                       //first one always counts
as do others because a gets reset in other loop

        //Set the initial Eval* variables
        EvalTreelist        = HitList[a].Treelist;
        EvalGoal            = HitList[a].Goal;
        EvalHold            = HitList[a].Hold;
        EvalInterval        = HitList[a].Interval;


        //sine HitList is already sorted, start at next record and look downward until no longer a match
        for(b=a+1;b<Count;)
        {
                if(     HitList[b].Treelist == EvalTreelist &&
                        HitList[b].Goal             == EvalGoal &&
                        HitList[b].Hold             == EvalHold &&
                        HitList[b].Interval == EvalInterval     )

                        b++;
        //look at next record
                else
                {
                        //Set the "a" variable to where "b" is because this is the next unique match
                        a = b;
                        break;
                }
        }//end for(b=a+1;b<Count;b++)
}//end for(a=0;a<Count;a++)

return Unique;
}//end CountUniqueFireHits


//*************************************************************************************
int CompareHitListForFlame(const void *ptr1, const void *ptr2)
//*************************************************************************************
{

        //Just to typecast them since we aren't actually passing in pointers
        struct HIT_BY_DISTURB *elem1;
        struct HIT_BY_DISTURB *elem2;

        elem1 = (struct HIT_BY_DISTURB *)ptr1;
        elem2 = (struct HIT_BY_DISTURB *)ptr2;
```

```
    if( elem1->Treelist < elem2->Treelist )
          //First sort by Treelist
        return -1;
    if( elem1->Treelist > elem2->Treelist )
        return 1;
    else
                                    //Then by Goal
          {
                    if( elem1->Goal < elem2->Goal )

                              return -1;
                    if( elem1->Goal > elem2->Goal )
                              return 1;
                    else
                                              //Then by Hold
                    {
                              if( elem1->Hold < elem2->Hold )

                                        return -1;
                              if( elem1->Hold > elem2->Hold )
                                        return 1;
                              else
                                              //Then by flame Interval
                              {
                                        if( elem1->Interval < elem2->Interval )

                                              return -1;
                                        if( elem1->Interval > elem2->Interval )
                                              return 1;
                                        else

                                              return 0;
                                        //FINISHED!!
                              }//end Interval
                    }//end Hold
          }//end Goal

}//end CompareHitListForFire

//*****************************************************************************************
int FillFireHitList(struct HIT_BY_DISTURB HitList[], int Per)
//*****************************************************************************************
{
/*
Once HitList has been created in ApplyFireDisturbance, this function will fill it up.  This
is pretty much a copy of CountFireHit, except this time variables will be stored in the HitList
structures for those cells that are hit.

Because CountFireHit() creates the new flame.asc file with flame length values in FEET , this function will now
call up a function to put that value into 2' interval values (which is what the FOFEM matrix has effects for).
*/
//Some string arrays
char garbage[13];
char FlameFile[250];

//File pointers
FILE *READ_FLAME;

//pointers
int *ptr_link;
ulong *ptr_treelist, *ptr_cellid;
ushort *ptr_gridcolumn, *ptr_goal, *ptr_hold;

//Misc. variables
int Row,Column;
int r,c,HowMany,ctr;
long int Nodata;
double xll, yll;
double TestValue, junk;
int Record, Interval;
//---------------------------- end of variable defining -------------------------------------------

//Make the flame file name and open it
sprintf(FlameFile, "%s%s%d\\per%d\\flame.asc",PREFIX,OUTPUTS,GOAL_TO_USE,Per);
READ_FLAME = fopen(FlameFile,"r");


//Read in the header info from the flame.asc file to get to the Real Data!
//This assumes the data was check for errors in CountFireHit()
fscanf(READ_FLAME,"%s %d %s %d %s %lf %s %lf %s %lf %s %ld",
                                        garbage, &Column, garbage, &Row, garbage, &xll, garbage, &yll,
                                        garbage, &junk, garbage, &Nodata);


//Scan in the values from flame.asc. If they are valid (not Nodata nor NONFOREST) then include them.  REMEMBER, if
//Nodata exists in the Data.GridColumn[] array, then there was originally nodata for this cell.
Record = 0;
for(r=1;r<=ROWS;r++)
{
          ptr_link = &link[r-1][1];
          HowMany = *(ptr_link+1);

          for(c=1;c<=COLUMNS;c++)
```

```
        {
                fscanf(READ_FLAME,"%lf", &TestValue);

                if(TestValue != Nodata)                                         //YES, it is a valid
number
                {

                        ptr_gridcolumn      = &Data.GridColumn[(*ptr_link)-1];      //set pointers
                        ptr_treelist        = &Data.Treelist[(*ptr_link)-1];
                        ptr_goal            = &Data.Goal[(*ptr_link)-1];
                        ptr_hold            = &Data.Hold[(*ptr_link)-1];
                        ptr_cellid                  = &Data.Cellid[(*ptr_link)-1];

                        for(ctr=0;ctr<HowMany;ctr++)
                        {
                                if(*ptr_gridcolumn == (ushort)c)        //FOUND the correct column
                                {                       //If a common GridColumn is not found - then Nodata
existed in orig Data.*[] arrays
                                        if(*ptr_treelist != NONFOREST && TestValue != FALSE)
        //Must be a valid NONFOREST cell
                                        {
                                                Interval = FlameInterval( (int)TestValue );

                                                //Now store all the needed data in the array of HitList
structures
                                                HitList[Record].Treelist      = *ptr_treelist;
                                                HitList[Record].Goal                  = *ptr_goal;
                                                HitList[Record].Hold                  = *ptr_hold;
                                                HitList[Record].Interval      = Interval;
                                                HitList[Record].Cellid                = *ptr_cellid;

                                                Record++;
                        //To send back as a counter
                                        }
                                        break;                                          //get out of
this for(ctr=0;ctr<HowMany...loop
                                }
                                ptr_gridcolumn++;
                                ptr_treelist++;
                                ptr_goal++;
                                ptr_hold++;
                                ptr_cellid++;
                        }//end of            for(ctr=0;ctr<HowMany;ctr++)
                }
        }
}//end of for(r=1;r<=ROWS;r++)
fclose(READ_FLAME);

return Record;
}

// ************************************************************
int CountFireHit(int per)
// ************************************************************
{
/*
After a run of Farsite, it will create a file called "per*_flame.grd", which I will
copy over to the correct "\\per*\\" directory first.

If the above file does not exist then either there were no fires that period
or the fire size was so small that FARSITE did not create an output flame.grd.  In any case,
this function will be skipped if there is no per*_flame.grd file available.

Otherwise, this function will go through the output flame grid file and count up how
many cells were actually hit by fire.  If, by chance, the fire occurs in a cell where there is
no data in the Data.* arrays, then it will be skipped with no repercussions (i.e. not counted).

Also, NONFOREST will be skipped

25 FEB 00:  Now will temporarily read the original Farsite generated grid and make
a new copy of it using the same strategy that was done in InOutFlammapResults().  The Farsite
generated file has -1 as the NoData value and ArcInfo seems to not like that.
*/
//-------------------------------------------------------------------------------------------------


//Some string arrays
char garbage[50];
char FlameFile[250];
char GrdFlameFile[250];
char SystemCall1[300];
char SystemCall2[250];
char SystemCall3[250];
char SystemCall4[250];

//File pointers
FILE *READ_FLAME;
FILE *WriteFlame;

//pointers
int *ptr_link;
ulong *ptr_treelist;
ushort *ptr_gridcolumn;

//Misc. variables
```

```
int Row,Column;
int r,c,HowMany,ctr;
double Nodata;
double xll, yll;
double TestValue, junk;
int CellsHit=0;

long CellTestValue;
ushort ConvertTest;
ushort *ptr_farsite;

FILE *BIN;
char InFile[256];
long CellidND;

//Variable for writing the output files
int *ptr_srp;                  //Starting Row Position
ushort   *ptr_column;
int ColumnsLeft;
ushort StartColumn,OutColumn;

//---------------------------------------- End of variable defining ----------------------------------------

//Make the correct file names
sprintf(FlameFile, "%s%s%d\\per%d\\flame.asc",PREFIX,OUTPUTS,GOAL_TO_USE,per);
sprintf(GrdFlameFile, "%s%s%d\\per%d\\flame.grd",PREFIX,OUTPUTS,GOAL_TO_USE,per);
sprintf(SystemCall1, "del %s", FlameFile);
sprintf(SystemCall2, "move %s%s\\per%d_flame.grd %s%s%d\\per%d\\flame.grd",PREFIX,RasterOutDir,per,
PREFIX,OUTPUTS,GOAL_TO_USE,per);
sprintf(SystemCall3, "del %s%s\\per%d_arrive.grd",PREFIX,RasterOutDir,per);
sprintf(SystemCall4, "del %s", GrdFlameFile);

//Execute some system calls
system(SystemCall1);
system(SystemCall4);
system(SystemCall2);
system(SystemCall3);

//=============================================================================================================
======
//Check existence of valid output FLAME grid from FARSITE
//Open up the flame.grd file (to read)
READ_FLAME = fopen(GrdFlameFile,"r");
if (READ_FLAME == NULL)
{
          printf("!!!!!  There were no fires this period - skipping FireEffects !!!!!!\n");
          fclose(READ_FLAME);
          return FALSE;
}


printf("  Preparing to import and export the FARSITE flame heights (import in meters, export in FEET units)\n");
puts("~-----------------------------------------------------");


//Create and initialize the FarsiteValue array, which will get filled with Farsite values using Cellid.bin as a
guide
//to indicate those cells which were originally NODATA (because of nodata problem with Farsite).
ushort (*FarsiteValue) = new ushort [UNIQUE];
if(FarsiteValue == NULL)
                    printf("There was NOT enough memory for FarsiteValue with %lu elements\n",UNIQUE);
memset( FarsiteValue, 0, sizeof(FarsiteValue[0]) * UNIQUE);

// ===================== OPEN AND READ THE CELLID DATA (again) ========================================

//Create a temporary array to store the input Cellid binary data, which has data for every cell
float (*TempCellid)[COLUMNS] = new float[ROWS][COLUMNS];   //ROWS*COLUMNS is how many elements are in the initial
grid/binary file
if (TempCellid == NULL)
          printf("There was NOT enough memory for TempCellid with %lu elements\n",ROWS*COLUMNS);

//Initialize the TempCellid array
memset( TempCellid, 0, sizeof(TempCellid[0][0]) * ROWS * COLUMNS);

//Check the header data associated with this binary file and get the returned NODATA value
CellidND = CheckHeader(0);

//******read in every element of the Cellid data and store in the TempCellid array
sprintf(InFile, "%s%s\\cellid_%s.bin",PREFIX,ConstantInput,ENVT);
BIN = fopen(InFile, "rb");
if( fread(TempCellid,sizeof(TempCellid),ROWS*COLUMNS,BIN) != ROWS*COLUMNS)      //TempCellid is only a pointer!!
          Bailout(66);
else
          printf("**Binary file %s OK**\n",InFile);
fclose(BIN);


// ================== READ THE FARSITE.GRD HEADER DATA - opened & checked existence earlier
============================
//Read in the header info from the flame.grd file to get to the Real Data!
fscanf(READ_FLAME,"%s %d %s %d %s %lf %s %lf %s %lf %s %lf",
                                                garbage, &Column, garbage, &Row, garbage, &xll, garbage, &yll,
                                                garbage, &junk, garbage, &Nodata);

//Do some error checking and bail if input data is not correct
```

```
if(Column != COLUMNS && Row != ROWS)
        Bailout(42);
if( floor(xll) != XLL && floor(yll) != YLL)
        Bailout(43);
// ==================== End of reading header data for files
=============================================================


//Scan in the values from flame.grd. If they are valid (not Nodata nor NONFOREST) then count them.  REMEMBER, if
//Nodata exists in the Data.GridColumn[] array, then there was originally nodata for this cell, so DO NOT count.
for(r=1;r<=ROWS;r++)
{
        ptr_link = &link[r-1][1];
        HowMany = *(ptr_link+1);

        for(c=1;c<=COLUMNS;c++)
        {
                //Use the TempCellid[][] as the "template" - if it has a value, then input
                //the data found in flame.grd - if there is one(after converting to the closest feet value)

                //First scan everyone in - one at a time so they all are on the same cell
                CellTestValue = (long)TempCellid[r-1][c-1];
                fscanf(READ_FLAME,"%lf", &TestValue);

                if(CellTestValue != CellidND)                           //This is a VALID cell
                {

                        if( TestValue == Nodata )     //not hit by fire
                                ConvertTest = 0;              //give it a zero flame length
                        else
                                ConvertTest = (ushort)( floor((TestValue*M2FT) + .5));      //convert to
closest ushort value

                        //Set pointer where this grid row starts in the Data.* array and in the FarsiteValue
array
                        ptr_gridcolumn = &Data.GridColumn[(*ptr_link)-1]; //set pointers
                        ptr_treelist   = &Data.Treelist[(*ptr_link)-1];
                        ptr_farsite =        &FarsiteValue[(*ptr_link)-1];


                        //look for this specific GridColumn in the Data.GridColumn array
                        for(Ctr=0;ctr<HowMany;ctr++)
                        {
                                if(*ptr_gridcolumn == (ushort)c)        //FOUND the correct column
                                {                       //If a common GridColumn is not found - then Nodata
existed in orig Data.*[] arrays

                                        *ptr_farsite = ConvertTest;

                                        if( *ptr_treelist != NONFOREST)
        //Only "count" if is not a NONFOREST cell
                                        {
                                                if(ConvertTest > 0 )
                //don't count those that get rounded to 0 flame length
                                                        CellsHit++;
                                        //To send back as a counter
                                        }

                                        break;                                                  //get out of
this for(ctr=0;ctr<HowMany...loop
                                }

                                ptr_gridcolumn++;
                                ptr_treelist++;
                                ptr_farsite++;
                        }//end of        for(ctr=0;ctr<HowMany;ctr++)

                }//end if(CellTestValue != CellidND)
        }//end for(c=1;c<=COLUMNS;c++)
}//end of for(r=1;r<=ROWS;r++)

fclose(READ_FLAME);

//Delete the TempCellid array from free store
delete [] TempCellid;

//=============================================================================
// Delete the flame.grd file
system(SystemCall4);
//=============================================================================

//Ok, the data is now stored in FarsiteValue[], so just spit those values back out

//open up the files to write to
WriteFlame =                fopen(FlameFile, "w");
if (WriteFlame == NULL)
        fprintf(stderr, "opening of %s failed: %s\n", FlameFile, strerror(errno));

//write out the header data
fprintf(WriteFlame,"ncols\t\t%d\n",COLUMNS);
fprintf(WriteFlame,"nrows\t\t%d\n",ROWS);
fprintf(WriteFlame,"xllcorner\t%.6lf\n",F_XLL);
fprintf(WriteFlame,"yllcorner\t%.6lf\n",F_YLL);
fprintf(WriteFlame,"cellsize\t%d\n",CELLSIZE);
fprintf(WriteFlame,"NODATA_value\t%d\n",NODATA);
```

```
for(r=1;r<=ROWS;r++)
{
          ptr_srp =                              &link[r-1][1];
          HowMany =                              *(ptr_srp+1);
          StartColumn =                          Data.GridColumn[(*ptr_srp)-1];          //not a pointer!
          ptr_column =                           &Data.GridColumn[(*ptr_srp)-1];
          ptr_farsite =                          &FarsiteValue[(*ptr_srp)-1];

          //If the whole row is blank, print out NODATA and goto next row
          if( *ptr_srp == FALSE )                //means a zero was left in this spot during MakeLink
          {
                    for(c=1;c<=COLUMNS;c++)
                              fprintf(WriteFlame,"%d  ",NODATA);

                    //put in new lines
                    fprintf(WriteFlame, "\n");

                    continue;           //goto next row
          }


          //print out NODATA for those cells before data starts
          for(c=1;c<StartColumn;c++)
                    fprintf(WriteFlame,"%d  ",NODATA);

          //set some counters
          OutColumn = StartColumn;
          ctr = 0;

          //print out values for area on landscape by checking
          //value in Data.GridColumn to match it with OutColumn value
          do{
                    if(*ptr_column == OutColumn)
                    {
                              if(*ptr_farsite == 0 )
                                        fprintf(WriteFlame, "%d  ",NODATA);
                              else
                                        fprintf(WriteFlame, "%hu  ",*ptr_farsite);

                              ptr_farsite++;
                              ptr_column++;
                              OutColumn++;
                              ctr++;
                    }
                    else      //print out NODATA for the "gaps"
                    {
                              fprintf(WriteFlame,"%d  ",NODATA);
                              OutColumn++;
                    }
          }while(ctr != HowMany );

          //Check to see how many columns are left to do
          ColumnsLeft = COLUMNS - (OutColumn-1);

          if(ColumnsLeft == 0)
          {
                    fprintf(WriteFlame, "\n");
                    continue;                       //go to next row
          }

          //print out NODATA for those cells after the data that are left
          for(c=0;c<ColumnsLeft;c++)
                              fprintf(WriteFlame,"%d  ",NODATA);

          //put in a new line
          fprintf(WriteFlame, "\n");

}//end of for(r=1;r<=ROWS;r++)


fclose(WriteFlame);

//Delete the FarsiteValue array on free store
delete [] FarsiteValue;


return CellsHit;
}//end of CountFireHit

// ****************************************************************
int FlameInterval(int f_feet)
// ****************************************************************
{

          //Will return an integer value that indicates what the  2' flame height group is.
          //The return value is the upper group height, so a FlameFeet of >=0 to 2 = 2, 3-4 =4, etc.

int group;
double INTERVAL = 2;
double Divide = 0;

          if(f_feet <=2)
                    return 2;
```

```
          Divide = (f_feet / INTERVAL);

          group = ( (int)(ceil(Divide)) * (int)INTERVAL;

          return group;
}//end of FlameInterval



//*****************************************************************************************************************
void DoubleCheckVegcodes(void)
//*****************************************************************************************************************
{
/*
Go through the entire Data.*[] arrays and make sure every cell has valid vegcode values
*/
int a,b;
ushort TempCode;
int TempCover,TempDiam,TempVeg;
//------------------------------- End of variable defining ---------------------------------------------------

printf("Getting ready to double check Vegcodes\n");

for(a=0;a<UNIQUE;a++)
{

          if(Data.Cellid[a] == FALSE)
                    break;

          if(Data.Treelist[a] == NONFOREST)
          {
                    for(b=0;b<NP;b++)
                    {
                              if(Data.Vegcode[a][b] != NONFOREST)
                                        printf("NONFOREST has a bad vegcode!\n");
                    }
          }
          else
          {

                    for(b=0;b<NP;b++)
                    {
                              TempCode = Data.Vegcode[a][b];                                        //The actual 3
or digit code from PREMO

                              if( TempCode > 1061 )
                                        printf("Problem with total TempCode\n");

                              //extract the digits out
                              TempCover = TempCode%10;
          //last digit for determining stage (is closure, <=60% or > 60% )
                              TempDiam = ( (TempCode-TempCover)%100 ) / 10;     //next to last digit also for
determining stage (is the QMD group)
                              TempVeg = (TempCode-TempCode%100) / 100;                  //lst  digit for
determining VegCode

                              if(TempCover > 1 )
                                        printf("Problem with COVER value\n");

                              if(TempDiam > 6 )
                                        printf("Problem with DIAM value\n");

                              if(TempVeg > 10 )
                                        printf("Problem with VEG value\n");
                    }

          }


}//end for(a=0 ...)

}//end DoubleCheckVegcodes
```

GOAL_CONTROLLER.CPP

```
/*
*****************************************************************************************************
 This GOAL_CONTROLLER.CPP file will hold the functions that are used for the landscape
                                                                                          optimization.

This file will hold the "PARENT" function that calls up the particular functions needed for particular landscape
goals we want to run.

Also, some functions that are fairly common to any goal and any heuristic are in here.

All heuristics used should employ the strategy of creating an array SOLUTION structures that has "X" records ;
where
"X" is a dynamic number reflecting the numbers of cells being evaluated and the 4 columns are:
    -cellid-treelist-goal-holdfor-          This format can be used for any type of spatial unit such as
subwatershed or for the entire basin.
```

```
*************************************************************************************************************
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"

//Functions declared here in goal_controller.cpp

void SwaplAdjust( struct SOLUTION *ptr_cs, ushort NG, ushort NH, double PerValues[], struct OPTIMIZE_SINGLE_VALUE
OV[],
                                    ulong Records, struct ERA *ptr_era);
void AsciiSaveGoalHold(void);
void BinarySaveGoalHold(void);
ulong CountSolutionWatersheds(ulong count, struct SOLUTION Solution[]);
int DecreaseShort(ulong count, ushort Short[][GOALS][HOLDNO]);
int DetermineEligibleCells(ulong Values[]);
int Fill_PValues(ulong Count, struct SOLUTION CS[], ulong Records, struct OPTIMIZE_SINGLE_VALUE OV[], double
Value[] );
int Fill_SEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[] );
int FillEndingEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[] );

int FillSolution(ulong Values[], struct SOLUTION Solution[], int Status);
int FillValueToOptimize(void);
double GetBaselineVTO(ulong count, struct SOLUTION Solution[], double PerValues[], struct OPTIMIZE_SINGLE_VALUE
OV[], ulong Records);
double GetSumBigTrees(ulong count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[], ulong Records);

int InputSolution(ulong Count, struct SOLUTION CS[] );
void InputAndCalculateSolutionEras(ulong Count, struct SOLUTION CS[] );
int LookAtOSV(const void *ptr1, const void *ptr2);
int CompareEraMinor(const void *ptr1, const void *ptr2);
int LookAtSolutionMinor(const void *ptr1, const void *ptr2);
int LookAtSolutionCellid(const void *ptr1, const void *ptr2);
void PickPrescriptions(int goal);
void ReuseBestPrescription(int goal);
void PrintSolutionValues(ulong Count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[],
                                                    ulong Records, int SubEra[], int Status);
void PrintSolutionBigTrees(ulong Count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[],
                                                    ulong Records, int SubEra[], int Status);
void PrintSolutionEraValues(struct ERA Era[], ulong NoSheds, int Status);

//functions to manage for goals (in goal*.cpp)
extern void Goal1(void);
extern void Goal3(void);
extern void Goal4(void);
extern void ReuseGoal(int Goal);

//Function is used here first but was originally coded during FillPremoData(), so it is found in PremoStuff.cpp
extern int LookAtPremoRecords(const void *ptr1, const void *ptr2);

//declared in EraStuff.cpp
extern void CalculateNetEras( struct CURRENT_ERAS *CellEra );
extern void CalculateDecayOnlyNetEras( struct CURRENT_ERAS *CellEra );

//Set in PremoStuff.cpp
extern ulong ITL;


//*************************************************************************************
void PickPrescriptions(int goal)
//*************************************************************************************
{

/*                                  =======  "goal" meanings: =======
1 = Max. Big Trees over entire watershed with subwatershed ERA constraint and w/thinning only of those stands <15"
in reserves
2 = Grow Only
*/

if(goal == 1)
{
        puts("\n\t\t*************************************************************************************************
******");
        printf("\t\t***                                GOAL #1
*****\n");
        printf("\t\t*** Getting ready to pick prescriptions based on Maximizing the # of Big Trees over the
*****\n");
        printf("\t\t*** entire watershed while constrained to a 6th-field subwatershed ERA threshold.
*****\n");
        printf("\t\t*** And with CONSERVATIVE management in the reserves (thinning only in stands <= 15\")
*****\n");
        printf("\t\t*** Both Federal and Private lands are eligible and included in this simulation.
*****\n");
        puts("\t\t*************************************************************************************************
****\n\n");

        Goal1();
}
else if(goal == GROW_ONLY)
{
```

```
        puts("\n*****************************************************************************************************
***");
        printf("****                                    GOAL #2
*****\n");
        printf("**** This is a GROW-ONLY scenario.  All stands will be assigned a StandGoal of grow-only with
*****\n");
        printf("****    a HOLDFOR value of 0.  Periodic disturbance will still be accounted for during growth.
*****\n");
        puts("*****************************************************************************************************
\n\n");

        ReuseGoal(goal);            //Just use the ReuseGoal() function

}
else if(goal == FINNEY_EFFECT)
{
        puts("\n*****************************************************************************************************
***");
        printf("****                                    GOAL #3
*****\n");
        printf("**** This is the FINNEY EFFECT scenario.  All the \"Bricks\" on Federal lands were assigned the
*****\n");
        printf("**** the Reduce Fire Hazard stand prescription.  All private lands were assigned the provide
*****\n");
        printf("**** a positive PNV stand prescription - all other cells were assigned Grow Only.
*****\n");
        puts("*****************************************************************************************************
\n\n");

        Goal3();
}
else if(goal == RX6)
{
        puts("\n*****************************************************************************************************
***");
        printf("****                                    GOAL #4
*****\n");
        printf("**** This goal is designed to mimic alternative 6 of the Framework draft alternatives.
*****\n");
        printf("**** All private lands were assigned the Provide Positve PNV goal.  Federal lands were
*****\n");
        printf("**** assigned stand prescriptions based on maximizing the the # of Big Trees over
*****\n");
        printf("**** the entire watershed while constrained to a 6th-field subwatershed ERA threshold and
*****\n");
        printf("**** some limitations on which prescriptions are allowed in particular areas.
*****\n");
        puts("*****************************************************************************************************
\n\n");

        Goal4();
}
else
        printf("\nNo optimization routine developed for that goal yet\n");


//Save the Goal-HoldFor values for entire landscape so I can reuse when running multiple simulations and want same
data
if(FILE_TYPE == 1)
AsciiSaveGoalHold();
else
BinarySaveGoalHold();


}//end PickPrescription

//*************************************************************************
void ReuseBestPrescription(int goal)
//*************************************************************************
{

/*                                  =======  "goal" meanings: =======
1 = Max. Big Trees over entire watershed with subwatershed ERA constraint and w/thinning only of those stands <15"
in reserves
2 = Grow Only
*/

if(goal == 1)
{
        puts("\n\t\t*****************************************************************************************************
******");
        printf("\t\t***                                    GOAL #1
*****\n");
        printf("\t\t*** Prescriptions were selected based on Maximizing the # of Big Trees over the
*****\n");
        printf("\t\t*** entire watershed while constrained to a 6th-field subwatershed ERA threshold.
*****\n");
        printf("\t\t*** And with CONSERVATIVE management in the reserves (thinning only in stands <= 15\")
*****\n");
        printf("\t\t*** Both Federal and Private lands are eligible and included in this simulation.
*****\n");
        puts("\t\t*****************************************************************************************************
****\n\n");

        ReuseGoal(goal);
```

```
}
else if(goal == GROW_ONLY)
{
          puts("\n*************************************************************************************************
**");
          printf("***                                    GOAL #2
*****\n");
          printf("**** This is a GROW-ONLY scenario.  All stands will be assigned a StandGoal of grow-only with
*****\n");
          printf("****    a HOLDFOR value of 0.  Periodic disturbance will still be accounted for during growth.
*****\n");
          puts("*************************************************************************************************
\n\n");

          ReuseGoal(goal);

}
else if(goal == FINNEY_EFFECT)
{
          puts("\n*************************************************************************************************
***");
          printf("****                                   GOAL #3
*****\n");
          printf("**** This is the FINNEY EFFECT scenario.  All the \"Bricks\" on Federal lands were assigned the
*****\n");
          printf("****    the Reduce Fire Hazard stand prescription - all other cells were assigned Grow Only.
*****\n");
          puts("*************************************************************************************************
\n\n");

          ReuseGoal(goal);

}
else if(goal == RX6)
{
          puts("\n*************************************************************************************************
**");
          printf("****                                   GOAL #4
*****\n");
          printf("**** This goal is designed to mimic alternative 6 of the Framework draft alternatives.
*****\n");
          printf("**** All private lands were assigned the Provide Positve PNV goal.  Federal lands were
*****\n");
          printf("**** assigned stand prescriptions based on maximizing the the # of Big Trees over
*****\n");
          printf("**** the entire watershed while constrained to a 6th-field subwatershed ERA threshold and
*****\n");
          printf("**** some limitations on which prescriptions are allowed in particular areas.
*****\n");
          puts("*************************************************************************************************
\n\n");

          ReuseGoal(goal);

}
else
          printf("\nNo optimization routine developed for that goal yet\n");

}//end ReuseBestPrescription

//*************************************************************************************************************
*****************
int Fill_SEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[] )
//*************************************************************************************************************
*****************
{
/*
This function is designed to fill the S_Era[] structures with the initial subwatershed cumulative ERA value for all
of those
subwatershed that are actually in the solution.  Since the Equivalent Roaded Acre (ERA) value is suppose to be a
cumulative measurement, this function will ignore whether or not particular cells are included in the solution -
just whether or
not a cell is in a subwatershed that is in the solution.

In the draft document, "Eldorado National Forest:  Cumulative Off-Site Watershed Effects (CWE) Analysis Process"
version 1.1
dated June, 1993, there was reference that they DID NOT include the acreages of wilderness in their ERA
calculations, but
after discussion with John Sessions we felt that we WILL include those acreages because the ERA is a cumulative
measurement.
However, this code could be modified to not count those acres if that is determined so later.

NOTE:  Both CS[] and S_Era[] MUST be coming in sorted by Minor in ascending order
*/

int a, Number;
ushort CurrentValue;
struct ERA Key;
struct ERA *ptr_record;

//----------------------------End of variable defining --------------------------------
//printf("Here in Fill_SEra\n");
//****************  First thing is to grab all the unique subwatershed ID's from CS[] ***************************

//Set CurrentValue to the first sub-watershed ID in CS
```

```
CurrentValue = CS[0].Minor;
S_Era[0].Minor = CurrentValue;
Number = 1;
//printf("Counting sub-watershed %hu\n",CS[0].Minor);

for(a=0;a<(signed)Count;a++)
{
          if(CS[a].Minor == CurrentValue)
                    continue;                                                                    //don't count
          else
          {
                    //printf("Counting sub-watershed %hu\n",CS[a].Minor);
                    S_Era[Number].Minor          = CS[a].Minor;
                    CurrentValue                          = CS[a].Minor;
                    Number++;
          }
}

printf("In FillS_Era, there were %d subwatersheds counted - out of %d, Everything is ",Number, NoSub);

if(Number != (signed)NoSub )
{
          printf(" Not OK..bailing\n",Number);
          Bailout(85);
          return FALSE;
}
else
          printf(" OK...continuing\n\n");


//Since CS[] was sorted by Minor  prior to coming to this function, I will assume that the Minor
//values that are now in S_Era[].Minor are in ascending order and can be BSEARCH with no problems.

//************************** Now sum up the initial ERA's for each subwatershed in solution ********************

for(a=0;a<UNIQUE;a++)
{
          if( Data.Cellid[a] == FALSE )          //No more cells to check
                    break;

          //Since there are no restrictions such as not counting wilderness, every cell has a contribution to
cumulative ERA
          //as long as its subwatershed is in the solution.  Make a key with the subwatershed ID and search for it
          Key.Minor = Data.Minor[a];

          //Use bsearch to find the matching subwatershed in the array of Era structures
          ptr_record = (struct ERA*)bsearch(
                    &Key,
                    (void *)S_Era,
                    (size_t)NoSub,
                    sizeof( struct ERA),
                    CompareEraMinor );

          //A subwatershed may not be in the solution so this is difficult to say something is wrong - will assume
that a NULL
          //pointer only happens for subwatersheds not in solution and thus will skip to next cell (e.g. water
bodies are never in solution)
          if( ptr_record == NULL )
                    continue;

          //at this point we have a pointer at the proper S_Era structure and we have access to the
Data.InitialEra[] value.
          ptr_record->SumInitialEra     += Data.InitialEra[a];                    //Sum up the InitialEra[] for this
subwatershed
          ptr_record->Count                        ++;
          //and track how many total cells are being summed per subwatershed


}//end for(a=0 ... )

return TRUE;
}//end Fill_SEra

//*****************************************************************************************************************
****************
int FillEndingEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[] )
//*****************************************************************************************************************
****************
{
/*
NOTE:  Both CS[] and S_Era[] MUST be coming in sorted by Minor in ascending order

This function is basically a copy of Fill_SEra() except this only gets called at the end of a simulation.
The difference is that this functions wants to sum up the Data.Era[][] values for those
cells in the solution.
*/

int a, b, Number;
ushort CurrentValue;
struct ERA Key;
struct ERA *ptr_record;

//----------------------------End of variable defining ----------------------------
//***************  First thing is to grab all the unique subwatershed ID's from CS[] ****************************
```

```
//Set CurrentValue to the first sub-watershed ID in CS
CurrentValue = CS[0].Minor;
S_Era[0].Minor = CurrentValue;
Number = 1;
//printf("Counting sub-watershed %hu\n",CS[0].Minor);

for(a=0;a<(signed)Count;a++)
{
          if(CS[a].Minor == CurrentValue)
                    continue;                                                                    //don't count
          else
          {
                    //printf("Counting sub-watershed %hu\n",CS[a].Minor);
                    S_Era[Number].Minor           = CS[a].Minor;
                    CurrentValue                            = CS[a].Minor;
                    Number++;
          }
}

printf("In FillS_Era, there were %d subwatersheds counted - out of %d, Everything is ",Number, NoSub);

if(Number != (signed)NoSub )
{
          printf(" Not OK..bailing\n",Number);
          Bailout(85);
          return FALSE;
}
else
          printf(" OK...continuing\n\n");


//Since CS[] was sorted by Minor  prior to coming to this function, I will assume that the Minor
//values that are now in S_Era[].Minor are in ascending order and can be BSEARCH with no problems.

//************************* Now sum up the initial ERA's for each subwatershed in solution ********************

for(a=0;a<UNIQUE;a++)
{
          if( Data.Cellid[a] == FALSE )           //No more cells to check
                    break;

          //Since there are no restrictions such as not counting wilderness, every cell has a contribution to
cumulative ERA
          //as long as its subwatershed is in the solution.  Make a key with the subwatershed ID and search for it
          Key.Minor = Data.Minor[a];

          //Use bsearch to find the matching subwatershed in the array of Era structures
          ptr_record = (struct ERA*)bsearch(
                    &Key,
                    (void *)S_Era,
                    (size_t)NoSub,
                    sizeof( struct ERA),
                    CompareEraMinor );

          //A subwatershed may not be in the solution so this is difficult to say something is wrong - will assume
that a NULL
          //pointer only happens for subwatersheds not in solution and thus will skip to next cell (e.g. water
bodies are never in solution)
          if( ptr_record == NULL )
                    continue;

          //at this point we have a pointer at the proper S_Era structure and we have access to the
Data.InitialEra[] value.
          for(b=0;b<NP;b++)
                    ptr_record->SumPeriodEra[b]   += Data.Era[a][b];            //Sum up the periodic Data.Era[][]
valuesfor this subwatershed

          ptr_record->Count                       ++;
          //and track how many total cells are being summed per subwatershed


}//end for(a=0 ... )

return TRUE;
}//end FillEndingEra

//*****************************************************************************************************************
****************
int Fill_PValues(ulong Count, struct SOLUTION CS[], ulong Records, struct OPTIMIZE_SINGLE_VALUE OV[], double
Value[] )
//*****************************************************************************************************************
****************
{
/*
This function is designed to go through the current solution stored in CS[] and tally up the sum total value, for
all the
different prescriptions found.  This is done by making a key from CS and looking for that key in the OV[] structure
and
looking at the Value[] member in there.

NOTE: CS[] is sorted by Cellid in ascending order & OV[] sorted by TREELIST-GOAL-HOLD
*/

int a,c;
```

```
//structure stuff
struct OPTIMIZE_SINGLE_VALUE Key;
struct OPTIMIZE_SINGLE_VALUE *ptr_key;

//----------------------------End of variable defining -------------------------------

//=========== Now go through the CS[] structures and tally up the Optimizing Value for those cells in the solution
====================

for(a=0;a<(signed)Count;a++)
{

          //Now make a key to look up the Optimizing value for this particular stand prescription in the array of
OV structures.
          Key.Treelist        = CS[a].Treelist;
          Key.Goal            = CS[a].Goal;
          Key.Hold            = CS[a].Hold;

          //Now use bsearch to find the matching record in the array of OV structures
          ptr_key = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
                    &Key,
                    (void *)OV,
                    (size_t)Records,
                    sizeof( struct OPTIMIZE_SINGLE_VALUE),
                    LookAtOSV );

          if(ptr_key == NULL)                              //There had better be one!
                    Bailout(86);
          else
          {
                    //Track the Value being optimized which returns back to heuristic as PerValues[]
                    for(c=0;c<NP;c++)
                              Value[c]                       += ptr_key->Value[c];
          }

}//end for(a ...)

return TRUE;

}//end Fill_PValues


//*****************************************************************************************
ulong CountSolutionWatersheds(ulong count, struct SOLUTION Solution[])
//*****************************************************************************************
{
/*
This will count up return the number of sub-watersheds that are actually in the current
solution.  The difference with the CountSubWatersheds() (in Misc.cpp) is that here those
GIS slivers and water bodies that were considered sub-watersheds are not counted.

This function will assume that the array of Solution structures has been sorted by the
.Minor  member .
*/

int a, Number;
ushort CurrentValue;

//----------------------------End of variable defining -------------------------------

//Set CurrentValue to the first sub-watershed ID in Solution
CurrentValue = Solution[0].Minor;
Number = 1;
//printf("Counting sub-watershed %hu\n",CurrentValue);

for(a=0;a<(signed)count;a++)
{
          if(Solution[a].Minor == CurrentValue)
                    continue;                                                          //don't count
          else
          {
                    //printf("Counting sub-watershed %hu\n",Solution[a].Minor);
                    Number++;
                    CurrentValue = Solution[a].Minor;
          }
}

//Print stuff to the stats.txt file
PrintToStat(4, (ulong)Number);

return (ulong)Number;
}//end CountSolutionWatersheds

//*****************************************************************************************
int DetermineEligibleCells(ulong Values[])
//*****************************************************************************************
{
/*
Depending on the GOAL_TO_USE, this function will go through the landscape and determine
some values that will be placed in the Values[] array and can then be used to
dynamically allocate space in the SOLUTION structure that will be created later.

NOTE:  This function can be used for all goals and anytime in during simulation when
outputs are needed to mimic the starting conditions (e.g. to get the potential BigTrees
given the initial landscape and using different goal scenarios - in OutputPreSimData )
```

```
*/

int a;

ushort *ptr_minor, *ptr_alloc, *ptr_stage, *ptr_buffer, *ptr_owner;
ulong *ptr_treelist, *ptr_cellid;
ulong AllocOK=0, AllocNOK=0,CellsInShed=0;

//---------------------------------- End of variable defining ------------------------------

if(GOAL_TO_USE == 1)
{

          for(a=0;a<UNIQUE;a++)
          {
                    ptr_alloc =                   &Data.Alloc[a];
                    ptr_minor =                   &Data.Minor[a];
                    ptr_treelist =      &Data.Treelist[a];
                    ptr_stage =                   &Data.InitialStage[a];
                    ptr_buffer =        &Data.Buffer[a];
                    ptr_cellid =        &Data.Cellid[a];

                    if(*ptr_cellid == FALSE)
                              break;

                    CellsInShed++;

                    if( *ptr_minor == WATER_BODY || *ptr_minor == NODATAFLAG     ||                  //if a lake or
one of those sliver subwatersheds
                              *ptr_alloc == ALLOC_WILD || *ptr_treelist == NONFOREST       ||                  //if
Wilderness or Treelist 209 or
                              (*ptr_alloc == ALLOC_RESERVE && *ptr_stage > 9)                         ||
          //if in LSR and >15" QMD or
                              (*ptr_buffer == IN_BUFFER && *ptr_stage > 9) )
                    //if in BUFFER and >15" QMD...
                              AllocNOK++;
                                                  //These ARE NOT eligible
                    else
                              AllocOK++;
          }
}
else if(GOAL_TO_USE == GROW_ONLY)
{
          for(a=0;a<UNIQUE;a++)
          {
                    ptr_alloc =                   &Data.Alloc[a];
                    ptr_minor =                   &Data.Minor[a];
                    ptr_treelist =      &Data.Treelist[a];
                    ptr_cellid =        &Data.Cellid[a];

                    if(*ptr_cellid == FALSE)
                              break;

                    CellsInShed++;

                    if( *ptr_minor == WATER_BODY || *ptr_minor == NODATAFLAG     ||                  //if a lake or
one of those sliver subwatersheds,
                              *ptr_alloc == ALLOC_WILD || *ptr_treelist == NONFOREST                )         //if
Wilderness or NONFOREST
                              AllocNOK++;
                                                  //These ARE NOT eligible
                    else
                              AllocOK++;
          }
}
else if(GOAL_TO_USE == FINNEY_EFFECT)
{
          for(a=0;a<UNIQUE;a++)
          {
                    ptr_owner =                   &Data.Owner[a];
                    ptr_treelist =      &Data.Treelist[a];
                    ptr_cellid =        &Data.Cellid[a];

                    if(*ptr_cellid == FALSE)
                              break;

                    CellsInShed++;

                    if( *ptr_owner == OWN_PI || *ptr_treelist == NONFOREST )    //if private lands (both PI and
PNI) or nonforest
                              AllocNOK++;
                                                  //These ARE NOT eligible
                    else
                              AllocOK++;
          }
}
else if(GOAL_TO_USE == RX6)
{
          for(a=0;a<UNIQUE;a++)
          {
                    ptr_owner =                   &Data.Owner[a];
                    ptr_treelist =      &Data.Treelist[a];
                    ptr_cellid =        &Data.Cellid[a];

                    if(*ptr_cellid == FALSE)
```

```
                        break;

                CellsInShed++;

                if( *ptr_owner == OWN_PI || *ptr_treelist == NONFOREST )    //if private lands (both PI and
PNI) or nonforest
                        AllocNOK++;
                                                //These ARE NOT eligible
                else
                        AllocOK++;
        }
}



else
        return FALSE;


//Fill the Values array in this order:
Values[0] = AllocOK;
Values[1] = AllocNOK;
Values[2] = CellsInShed;


        return TRUE;
}//end DetermineEligibleCells



//*****************************************************************************
int FillSolution(ulong Values[], struct SOLUTION Solution[], int Status)
//*****************************************************************************
{
/*
This function fills up the array of SOLUTION structures for a goal.  What always gets
filled is the Minor, Cellid, and Treelist values.  When this function is called up with
Status == FAKE, then it is being called after the landscape optimization (i.e. during
OutputPre- or PostSimAnalysisData) and the Goal & Hold were already found so they will be
filled as well.

Also, for the Grow Only goal (Goal2), the Goal & Hold values can be used from what was
put in during initialization of the Data*.[] arrays - because Goal 9 and Hold 0 was used.
*/

int b;

ushort *ptr_minor, *ptr_alloc, *ptr_stage, *ptr_buffer, *ptr_goal, *ptr_hold, *ptr_owner, *ptr_rule;
ulong *ptr_treelist, *ptr_cellid;
ulong AllocOK=0, AllocNOK=0,EligibleCell, CellsInShed=0;

//------------------------------- End of variable defining ---------------------------------------------------
----

//Set some variables from the incoming Values array
AllocOK             = Values[0];
AllocNOK  = Values[1];
CellsInShed         = Values[2];

EligibleCell=0;
if(GOAL_TO_USE == 1)
{

        for(b=0;b<UNIQUE;b++)
        {
                ptr_cellid =        &Data.Cellid[b];                        //ptr_cellid has the cellid for
this cell
                ptr_treelist =      &Data.Treelist[b];                      //ptr_treelist has the treelist
value for this cell
                ptr_minor =                 &Data.Minor[b];
                ptr_alloc =                 &Data.Alloc[b];
                ptr_stage =                 &Data.InitialStage[b];
                ptr_buffer =        &Data.Buffer[b];
                ptr_goal =                  &Data.Goal[b];
                ptr_hold =                  &Data.Hold[b];

                if(*ptr_cellid == FALSE)
                {
                        if(EligibleCell != AllocOK)
                                Bailout(51);
                        else
                                break;                                                          //just break
out, no more valid cells in Data.* array
                }


                if( *ptr_minor == WATER_BODY || *ptr_minor == NODATAFLAG     ||                         //if
a lake or one of those sliver subwatersheds
                        *ptr_alloc == ALLOC_WILD || *ptr_treelist == NONFOREST      ||
        //if Wilderness or NonForest or
                        (*ptr_alloc == ALLOC_RESERVE && *ptr_stage > 9)                         ||
                //if in LSR and >15" QMD or
                        (*ptr_buffer == IN_BUFFER && *ptr_stage > 9) )
                        //if in BUFFER and >15" QMD...
```

```
                    continue;
                //look at next cell
                else
                {
                    Solution[EligibleCell].Minor            = *ptr_minor;
                    Solution[EligibleCell].Cellid           = *ptr_cellid;
                    Solution[EligibleCell].Treelist                 = *ptr_treelist;

                    //Put in the Goal and Hold values found during landscape optimization when reusing a
solution already found
                    if(Status == FAKE)
                    {
                        Solution[EligibleCell].Goal             = *ptr_goal;
                        Solution[EligibleCell].Hold             = *ptr_hold;
                    }

                    EligibleCell++;


                    if(EligibleCell == AllocOK)                 //done
                        break;
                }

        } //end for(b=0;b<UNIQUE;b++)
}
else if(GOAL_TO_USE == GROW_ONLY)
{

        for(b=0;b<UNIQUE;b++)
        {
            ptr_cellid =        &Data.Cellid[b];                //ptr_cellid has the cellid for
this cell
            ptr_treelist =      &Data.Treelist[b];              //ptr_treelist has the treelist
value for this cell
            ptr_minor =             &Data.Minor[b];
            ptr_alloc =             &Data.Alloc[b];
            ptr_goal =              &Data.Goal[b];

            if(*ptr_cellid == FALSE)
            {
                if(EligibleCell != AllocOK)
                    Bailout(51);
                else
                    break;                                      //just break
out, no more valid cells in Data.* array
            }


            if( *ptr_minor == WATER_BODY || *ptr_minor == NODATAFLAG    ||                  //if
a lake or one of those sliver subwatersheds
                *ptr_alloc == ALLOC_WILD || *ptr_treelist == NONFOREST          )
        //if Wilderness or NonForest or
                continue;
            //look at next cell
            else
            {
                Solution[EligibleCell].Minor            = *ptr_minor;
                Solution[EligibleCell].Cellid           = *ptr_cellid;
                Solution[EligibleCell].Treelist                 = *ptr_treelist;

                Solution[EligibleCell].Goal                     = *ptr_goal;        //For Grow
Only, the Goal & Hold were initialized with correct values
                Solution[EligibleCell].Hold                     = 0;

                //NOTE:  No need to have if(Status == FAKE) - the goal & hold get filled above no
matter what

                EligibleCell++;


                if(EligibleCell == AllocOK)                 //done
                    break;
            }

        } //end for(b=0;b<UNIQUE;b++)
}
else if(GOAL_TO_USE == FINNEY_EFFECT)
{

        for(b=0;b<UNIQUE;b++)
        {
            ptr_cellid =        &Data.Cellid[b];                //ptr_cellid has the cellid for
this cell
            ptr_treelist =      &Data.Treelist[b];              //ptr_treelist has the treelist
value for this cell
            ptr_minor =             &Data.Minor[b];
            ptr_owner =             &Data.Owner[b];
            ptr_rule =              &Data.PRule[b];
            ptr_goal =              &Data.Goal[b];

            if(*ptr_cellid == FALSE)
            {
                if(EligibleCell != AllocOK)
                    Bailout(51);
                else
```

```
                                        break;                                                  //just break
out, no more valid cells in Data.* array
                    }

                    //6 Mar 00: Klaus suggested giving all private lands the PNV stand goal prescription because
for the
                    //Framework those lands are PI and they are doing cutting, so this will at least simulate
something
                    if( *ptr_owner == OWN_PI || *ptr_treelist == NONFOREST )    //if private lands (both PI and
PNI) or nonforest
                    {
                              if( *ptr_owner == OWN_PI)
                                        *ptr_goal = SG_PNV;

                              continue;
                                        //look at next cell
                    }
                    else
                    {

                              Solution[EligibleCell].Minor            = *ptr_minor;
                              Solution[EligibleCell].Cellid           = *ptr_cellid;
                              Solution[EligibleCell].Treelist                   = *ptr_treelist;

                              /*
                              OK, this may seem weird - but Data.PRule[], had either a 1 or 10 value and that can
be used directly
                              as the stand goal for eligible cells.  However, they must have 1 subtracted from them
to maintain
                              the numbering system used for the stand goals.
                              */
                              Solution[EligibleCell].Goal                       = (*ptr_rule) - 1;
                              Solution[EligibleCell].Hold                       = 0;

                              //NOTE:  No need to have if(Status == FAKE) - the goal & hold get filled above no
matter what

                              EligibleCell++;

                              if(EligibleCell == AllocOK)                       //done
                                        break;
                    }
          }//end for(b=0 ... )
}
else if(GOAL_TO_USE == RX6)
{

          for(b=0;b<UNIQUE;b++)
          {
                    ptr_cellid =        &Data.Cellid[b];                        //ptr_cellid has the cellid for
this cell
                    ptr_treelist =      &Data.Treelist[b];                      //ptr_treelist has the treelist
value for this cell
                    ptr_minor =                 &Data.Minor[b];
                    ptr_owner =                 &Data.Owner[b];
                    ptr_rule =                  &Data.PRule[b];
                    ptr_goal =                  &Data.Goal[b];
                    ptr_hold =                  &Data.Hold[b];

                    if(*ptr_cellid == FALSE)
                    {
                              if(EligibleCell != AllocOK)
                                        Bailout(51);
                              else
                                        break;                                  //just break
out, no more valid cells in Data.* array
                    }

                    //6 Mar 00: Klaus suggested giving all private lands the PNV stand goal prescription because
for the
                    //Framework those lands are PI and they are doing cutting, so this will at least simulate
something
                    if( *ptr_owner == OWN_PI || *ptr_treelist == NONFOREST )    //if private lands (both PI and
PNI) or nonforest
                    {
                              if( *ptr_owner == OWN_PI)
                                        *ptr_goal = SG_PNV;

                              continue;
                                        //look at next cell
                    }
                    else
                    {

                              Solution[EligibleCell].Minor            = *ptr_minor;
                              Solution[EligibleCell].Cellid           = *ptr_cellid;
                              Solution[EligibleCell].Treelist                   = *ptr_treelist;

                              /*
                              OK, this may seem weird - but Data.PRule[], has values 1 thru 10 in it, which
represent the Maximum stand
                              goal prescription # that a cell can have (e.g. a value of 2 means it can have stand
goal 1 or 2).
                              However, they must have 1 subtracted from them to maintain the numbering system used
for the stand goals.
```

```
                                */
                                Solution[EligibleCell].MaxGoal                                  = (*ptr_rule) - 1;

                                //Put in the Goal and Hold values found during landscape optimization when reusing a
solution already found
                                if(Status == FAKE)
                                {
                                        Solution[EligibleCell].Goal             = *ptr_goal;
                                        Solution[EligibleCell].Hold             = *ptr_hold;
                                }

                                EligibleCell++;

                                if(EligibleCell == AllocOK)                     //done
                                        break;
                        }
                }//end for(b=0 ... )
}
else
        return FALSE;


        return TRUE;
}//end FillSolution


//*****************************************************************************
int FillValueToOptimize()
//*****************************************************************************
{
/*
NOTE:  Can be used for either a Subwatershed or Watershed search

This function will create a shortened version of the Premo data that was created
in the CreateSortedPremoBinaryFile() function.  That structure has a different
record for every Treelist-Goal-Hold-Period combination whereas this function will
create an array of structures that has Treelist-Goal-Hold-Value[NP]-BigTrees[NP]-Rev[NP], and CFHarvest[NP].
Only the Treelist-Goal-Hold differentiate new records.  The value that is placed
in the *[].Value[] spot is that value being OPTIMIZED.  So for example, in goal 1 it will be the #of
BigTrees. The value in *[].BigTrees[] will always be the #of BigTrees (so for goal 1, it will have the
same data as in *[].Value[])

NOTE:  The value stored will always be USHORT to help in reducing processing time.
Once a solution has been found, FillPremoData() will enter the real data as float or ushort.

Once this array of structures is completed and sorted, it will be written out to
a binary file to be used later on during the landscape optimization.  Could recode later to
pass a pointer to a structure but this is OK for now.
*/

//IO variables
FILE *BinIn, *HeaderIn, *Index, *BinOut, *HeaderOut;
char Temp[256];
ulong RecordNo;

//structures
struct PREMO_RECORD Key;
struct PREMO_RECORD *ptr_key;

int count, goal, Hold;
int ScanStatus,IndexNo, ctr;
ushort Per;
ulong Record;
ulong POT;

//For Time information
clock_t Start, Finish;
double Duration;

//for misc counting
int x=0, y=0;

//------------------------- end variable defining ------------------------------------


//calculate the global POT - this is how many actual structures (or records) are in the InitOpt array of structures
POT = ITL * GOALS * HOLDNO;
//printf("So I just set POT to value of %lu\n",POT);


//+++++++++++++++++++++ The "smaller" structure to hold value being optimized and BigTrees
++++++++++++++++++++++++++++
//declare and Initialize the array ofInitOpt[] structures - this is a compact version of the PInv structure further
below
struct OPTIMIZE_SINGLE_VALUE (*InitOpt) = new struct OPTIMIZE_SINGLE_VALUE[POT];
if(InitOpt == NULL)
        printf("Problems allocating memory for InitOpt with %lu records\n",POT);

memset(InitOpt, 0, sizeof(struct OPTIMIZE_SINGLE_VALUE) * POT);


//+++++++++++++++++++ The original structure that holds all the Premo data ++++++++++++++++++++++++++++++++++

//Read in the binary file created by CreateSortedPremoBinaryFile() - it has all the data for the initial stands in
it
```

```
//See PremoStuff.cpp - I originally created this process to read in the Premo data so more info is located there

//Create and Open the Header and actual Binary file with PREMO data in it
sprintf(Temp, "%s%s\\Binary\\%s_Premo.bin",PREFIX,InitialStandDataDir,ENVT);
BinIn = fopen(Temp, "rb");

sprintf(Temp, "%s%s\\Binary\\%s_Premo.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderIn = fopen(Temp, "r");

//Get the Number of records that are listed in the header file
fscanf(HeaderIn,"%lu",&RecordNo);

//Create an array of structures on the free store to hold these records
struct PREMO_RECORD (*PInv) = new struct PREMO_RECORD[RecordNo];                    //PInv stands for
Potential Inventory
if( PInv == NULL )
        printf("Problems allocating memory for PInv[] with %lu elements\n",RecordNo*sizeof(PREMO_RECORD));

//Initialize a couple of things
memset( PInv, 0, sizeof(struct PREMO_RECORD) * RecordNo );  //array of structures to hold all the input data
memset( &Key, 0, sizeof( struct PREMO_RECORD) );                    //Key to use for searching for a particular
record in PInv


//Now just read in the binary data the same way it was written out in CreateSortedPremoBinaryFile()
fread(PInv, sizeof(PREMO_RECORD),RecordNo,BinIn);

//close up the files
fclose(BinIn);
fclose(HeaderIn);

//Set an error checker to check the value of RecordNo
if(RecordNo != POT * NP  )
        Bailout(78);


//NOTE: PInv is sorted by:  Treelist - Goal - Hold - Period

//Create a shortened version of PInv by placing equivalent data in the array ofInitOpt structures.  That way
InitOpt can be sorted
//and there will be few records to BSEARCH through because the values to optimize on will ALL be stored in the
//InitOpt->Value[] array (which is accessible by finding only one record, not NP records!)

//I am assuming that a "...\treeindex.txt" file exists (made during InitialStandOpt() )
        sprintf(Temp, "%s%s%d\\per0\\%s",PREFIX,INPUTS,GOAL_TO_USE,TREE_INDEX);
        Index = fopen(Temp,"r");

        if (Index == NULL)
                fprintf(stderr, "opening of %s failed(FillValueToOptimize() ):  %s\n", Temp,strerror(errno));


        // First go through the file and COUNT the number of treeindexes
        count = 0;
        while ((ScanStatus=fscanf(Index,"%d",&IndexNo))!=EOF)
        {
                count = ++count;
        }

        // Rewind the file pointer so it is back at the beginning of the file
        rewind(Index);

//An error checker - these two should match
if(ITL != (unsigned)count)
        Bailout(79);

// Start looking at each initial treelist, and for each Goal and Hold combo, fill in the array of InitOpt
structures with
//the corresponding data in the array of PInv structures.
Record=0;
for(ctr = 0; ctr < count; ctr++)                                                    //for each treelist
{
        fscanf(Index,"%d",&IndexNo);                    //scan in the value - if this function used later,
watch out for treelist > 65,530

        for(goal=0;goal<GOALS;goal++)                                               //for each goal
        {
                for(Hold=0;Hold<4;Hold+=3)    //for the "HoldFor" periods
                {

                        //Start to fill in the array of InitOpt structures with the above data - this is an
OPTIMIZE_SINGLE_VALUE type
                        InitOpt[Record].Treelist      = (ulong)IndexNo;
                        InitOpt[Record].Goal                  = (ushort)goal;
                        InitOpt[Record].Hold                  = (ushort)Hold;

                        //Start to make a key for this combination of IndexNo - goal - Hold - Elev
                        //The key is a PREMO_RECORD type so it can look through the PInv structure
                        Key.Treelist        = (ulong)IndexNo;
                        Key.Goal            = (ushort)goal;
                        Key.Hold            = (ushort)Hold;

                        for(Per=0;Per<NP;Per++)
                        {
                                //Finish off the Key with the period
                                Key.Period                  = (ushort)Per;
```

```
                                        //Now use bsearch to find the matching record in the array of PInv
structures
                                        ptr_key = (struct PREMO_RECORD*)bsearch(
                                                &Key,
                                                (void *)PInv,
                                                (size_t)RecordNo,
                                                sizeof( struct PREMO_RECORD),
                                                LookAtPremoRecords  );


                                        if(ptr_key == NULL)
                                                Bailout(75);

                                        //********************************* actual Value to OPTIMIZE
*****************************
                                        //Fill in the current InitOpt[Record].Value[Per] with the correct value
from the record ptr_key points to
                                        if(GOAL_TO_USE == 1 )
                                                InitOpt[Record].Value[Per]    = (ushort)(floor( ptr_key->BigTrees
* ACREEQ));          //divide by BIGTREES_EXP when done
                                        else if(GOAL_TO_USE == GROW_ONLY)
                                                InitOpt[Record].Value[Per]    = 0;
                                        //Grow Only, no need to put anything here
                                        else if(GOAL_TO_USE == FINNEY_EFFECT)
                                                InitOpt[Record].Value[Per]    = 0;
                                        //using the "bricks", no need to put anything here
                                        else if(GOAL_TO_USE == RX6 )
                                                InitOpt[Record].Value[Per]    = (ushort)(floor( ptr_key->BigTrees
* ACREEQ));          //divide by BIGTREES_EXP when done
                                        else
                                                Bailout(62);

                                        //********************************* The # of Big Trees
************************************
                                        //And then fill InitOpt[Record].BigTrees[Per] with the #of BigTrees for
that record...NO matter what goal!
                                        InitOpt[Record].BigTrees[Per] =  (float)(ptr_key->BigTrees * ACREEQ);
          //need to divide by BIGTREES_EXP when done

                                        //********************************* The associated REVENUE
*******************
                                        //If the revenue is (-), then just make it 0 ...also make sure no values
over ushort get in
                                        if( ptr_key->Rev < 0 )
                                                InitOpt[Record].Rev[Per]                = (ushort)0;
                                        else if( ptr_key->Rev > 65530 )
                                                InitOpt[Record].Rev[Per]                = (ushort)65530;
                                        else
                                                InitOpt[Record].Rev[Per]                = (ushort)ptr_key->Rev;

                                        //********************************* The associated CFHarvest
*******************
                                        //If the harvest is (-), then just make it 0 (should never happen!) ...also
make sure no values over ushort get in
                                        if( ptr_key->Harvest < 0 )
                                                InitOpt[Record].CFHarvest[Per]                    = (ushort)0;
                                        else if( ptr_key->Harvest > 65530 )
                                                InitOpt[Record].CFHarvest[Per]                    =
(ushort)65530;
                                        else
                                                InitOpt[Record].CFHarvest[Per]                    =
(ushort)ptr_key->Harvest;


                                }//end for(Per=0;Per<NP;Per++)

                                Record++;                                                       //increment only when all
period values for one Treelist-Goal-Hold combo are entered

                        }//end of for(HoldPeriods...
                }//end of for(goal....
}               //end of for(ctr...

Start = clock();

//***** Sort the array of InitOpt structures by Treelist-Goal-Hold

qsort(  (void*)InitOpt,
          //base
        (size_t)POT,
          //count of records
                sizeof( struct OPTIMIZE_SINGLE_VALUE ),                                 //size of each record
                LookAtOSV );
                //compare function

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
//printf("***Finished sorting in %.2lf seconds\n",Duration);

//close the treeindex.txt file
fclose(Index);


//Create the output Binary file and header file
```

```
sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.bin",PREFIX,InitialStandDataDir,ENVT);
BinOut = fopen(Temp, "wb");

sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderOut = fopen(Temp, "w");

//Write out the header data -- need to know how many records there are
fprintf(HeaderOut, "%lu\n",POT);

//And now write out all the records in the array of Inv structure
fwrite(InitOpt, sizeof(OPTIMIZE_SINGLE_VALUE),POT,BinOut);

fclose(BinOut);
fclose(HeaderOut);


//delete any arrays on free store
delete [] InitOpt;
delete [] PInv;


          return TRUE;

} //end of FillValueToOptimize()


//*************************************************************************************************
int LookAtOSV(const void *ptr1, const void *ptr2)
//*************************************************************************************************
{
          //Will qsort or bsearch an OPTIMIZE_SINGLE_VALUE (OSV) type by Treelist-Goal-Hold

          //Just to typecast them since we aren't actually passing in pointers
          struct OPTIMIZE_SINGLE_VALUE *elem1;
          struct OPTIMIZE_SINGLE_VALUE *elem2;

          elem1 = (struct OPTIMIZE_SINGLE_VALUE *)ptr1;
          elem2 = (struct OPTIMIZE_SINGLE_VALUE *)ptr2;


    if( elem1->Treelist < elem2->Treelist )
          //First sort by Treelist
        return -1;
    if( elem1->Treelist > elem2->Treelist )
        return 1;
    else
                                        //Then by Goal
          {
                    if( elem1->Goal < elem2->Goal )

                              return -1;
                    if( elem1->Goal > elem2->Goal )
                              return 1;
                    else
                                                  //Then by Hold
                    {
                              if( elem1->Hold < elem2->Hold )

                                        return -1;
                              if( elem1->Hold > elem2->Hold )
                                        return 1;
                              else
                                        return 0;
                                        //FINISHED!.
                    }//end Hold
          }//end Goal

}//end LookAtOSV

//*************************************************************************************************
int LookAtSolutionMinor(const void *ptr1, const void *ptr2)
//*************************************************************************************************
{
          //Will qsort or bsearch an SOLUTION type by Minor in ascending order

          //Just to typecast them since we aren't actually passing in pointers
          struct SOLUTION *elem1;
          struct SOLUTION *elem2;

          elem1 = (struct SOLUTION *)ptr1;
          elem2 = (struct SOLUTION *)ptr2;


    if( elem1->Minor < elem2->Minor )
          //First by Minor
        return -1;
    if( elem1->Minor > elem2->Minor )
        return 1;
    else

                    return 0;
                                        //FINISHED!!

}//end LookAtSolutionMinor
```

```
//*****************************************************************************************
int LookAtSolutionCellid(const void *ptr1, const void *ptr2)
//*****************************************************************************************
{
          //Will qsort or bsearch an SOLUTION type by Cellid in ascending order

          //Just to typecast them since we aren't actually passing in pointers
          struct SOLUTION *elem1;
          struct SOLUTION *elem2;

          elem1 = (struct SOLUTION *)ptr1;
          elem2 = (struct SOLUTION *)ptr2;


    if( elem1->Cellid < elem2->Cellid )                                          //First by Cellid
        return -1;
    if( elem1->Cellid > elem2->Cellid )
        return 1;
    else

                    return 0;
                    //FINISHED!!

}//end LookAtSolutionCellid
//*****************************************************************************************
int CompareEraMinor(const void *ptr1, const void *ptr2)
//*****************************************************************************************
{
          //Will qsort or bsearch an ERA type by minor in ascending order

          //Just to typecast them since we aren't actually passing in pointers
          struct ERA *elem1;
          struct ERA *elem2;

          elem1 = (struct ERA *)ptr1;
          elem2 = (struct ERA *)ptr2;

    if( elem1->Minor < elem2->Minor )                                            //First sort by
Minor
        return -1;
    if( elem1->Minor > elem2->Minor )
        return 1;
    else

                                    return 0;
                                    //FINISHED!!


}//end LookAtEra

//*****************************************************************************************************************************
*****************
double GetBaselineVTO(ulong count, struct SOLUTION Solution[], double PerValues[], struct OPTIMIZE_SINGLE_VALUE
OV[], ulong Records)
//*****************************************************************************************************************************
*****************
{
/*
This function can be used for any goal that has a single value to optimize and actually
used FillValueToOptimize() (which create the array of structures currently being passed in as OV[]).
*/


ulong a;
double SumValue = 0;
double Baseline, ReturnBaseline;


// ------------------------------ End of variable defining ------------------------------

/*
int x,y;
printf("\n\n****  AND NOW the OV structures in GetBaselineVTO!!\n");
for(x=0;x<30;x++)
{
          printf("OV[%d]:\t%lu\t%hu\t%hu", x, OV[x].Treelist, OV[x].Goal, OV[x].Hold);
          for(y=0;y<NP;y++)
                    printf("\t%hu",OV[x].Value[y]);
          printf("\n");
}
printf("There are %lu records in the OV array\n",Records);

printf("\n\n**** AND NOW the Solution structures in GetBaselineVTO!!!!\n");
for(x=0;x<30;x++)
          printf("Solution[%d]:\t%hu\t%lu\t%lu\t%hu\t%hu\n", x, Solution[x].Minor, Solution[x].Cellid,
                                                                                    Solution[x].Treelist,
Solution[x].Goal, Solution[x].Hold);
*/


if(GOAL_TO_USE == 1)
{
          //Call up GetSumBigTrees() to help determine a baseline to use for Big Tree goals
          Baseline = GetSumBigTrees(count, Solution, OV, Records);
}
```

```
else
{
          //Sum up the PerValues values and get the SumValue to use in caluculating a single AvgValue that
represents
          //the total deviation of all periodic "values" from a constant level.
          for(a=0;a<NP;a++)
                    SumValue += PerValues[a];

          Baseline = (SumValue / NP);

}//end else(GOAL_TO_USE ==1)


//Make adjustments to the baseline as needed
ReturnBaseline = Baseline * BASE_ADJ;


return ReturnBaseline;
}//end GetBaselineVTO


//*****************************************************************************************************************
*****************
double GetSumBigTrees(ulong count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[], ulong Records)

//*****************************************************************************************************************
*****************
{
/*
The ideal of this function is to find the SUM number of BigTrees (for all cells) per period.
The SUM will include using: 1) a grow only scenario (goal 9), 2) a Reduce
Wildfire only scenario (goal 0), and 3) enhance Fish Habitat only scenario(goal 2) - adding
the # of trees, at cell acre equivalent / per period and dividing by 3.  This will give me
something of an "average SUM" which can then be used during the optimization.
*/
int a, b, c, y, goal, hold;

float TempPeriodTotals[NP], SumBigTrees[NP];
double LargestSum=0;

struct OPTIMIZE_SINGLE_VALUE Key;
struct OPTIMIZE_SINGLE_VALUE *ptr_key;

//------------------------------------------- End of variable defining ------------------------------------

//Initialize the TempPeriodTotals[] and the SumBigTrees[] arrays
for(a=0;a<NP;a++)
{
          TempPeriodTotals[a] = 0;
          SumBigTrees[a]                = 0;
}

//Create a temp array to keep track of those cells "already counted" - save processing time
ushort (*Counted) = new ushort[count];
if( Counted == NULL)
          printf("There was NOT enough memory for Counted with %lu elements\n",count);


for(a=0;a<(signed)count;a++)
          Counted[a] = 0;


//Go through CS[] and find treelist currently in this initial solution
for(a=0;a<(signed)count;a++)
{
          //First check and see if this Treelist has already been opened and accounted for
          if( Counted[a] == TRUE )                          //YES, is has been, continue on to next cell
                    continue;

          //Otherwise, start to make a key to look for this treelist in the OV structures
          Key.Treelist        = CS[a].Treelist;

          //Make an inner loop to go through each of 3 differnt goal scenarios that will be used in getting the
"average SUM" value
          for(b=0;b<3;b++)
          {
                    //Get the "goal" value
                    if(b == 0)
                              goal = 0;           //Reduce Wildfire stand goal
                    else if(b == 1)
                              goal = 2;           //Enhance Fish Habitat stand goal
                    else
                              goal = 9;           //Grow Only stand goal

                    //Get the "hold" value
                    do{
                              hold = ( rand() % 4);                    //this gives 0,1,2,and 3....but it has to be
either 0 or 3
                    }while(hold == 1 || hold == 2);


                    //Finish off the key
                    Key.Goal  = (ushort)goal;
                    Key.Hold  = (ushort)hold;
```

```
                    //Now use bsearch to find the matching record in the array of OV structures
                    ptr_key = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
                              &Key,
                              (void *)OV,
                              (size_t)Records,
                              sizeof( struct OPTIMIZE_SINGLE_VALUE),
                              LookAtOSV );

                    if(ptr_key == NULL)                              //There had better be one!
                              Bailout(80);
                    else                                                                  //Add the values for Big
Trees to the TempPeriodTotals[] (which has sum for the 3 goal scenarios)
                    {
                              for(c=0;c<NP;c++)
                                        TempPeriodTotals[c] += ptr_key->BigTrees[c] ;
                    }

          }//end for(b=0;b<3;b++)

          /*
          The TempPeriodTotals[] values can be reused for all solution cells that have the same treelist.  Scroll
          through CS[] and find those that do have same treelist and make a flag in the Counted array if so.
          */

          //First make contribution for this cell since it was first to have this treelist
          for(y=0;y<NP;y++)
                    SumBigTrees[y] += TempPeriodTotals[y];

          //Set a flag in the Counted array
          Counted[a] = TRUE;                                                  //YES, this cell has now been
accounted for

          //Now start looking through remaining cells in CS[]
          for(b=a+1;b<(signed)count;b++)                                                        //start looking
at next cell
          {
                    if( CS[b].Treelist == CS[a].Treelist )                    //YES, this cell does have the same
treelist
                    {
                              //Set an error checker - each treelist should be done once and this indicates a
second time
                              if( Counted[b] == TRUE )
                                        Bailout(74);

                              //otherwise, add this treelist's contribution again to account for this cell
                              for(y=0;y<NP;y++)
                                        SumBigTrees[y] += TempPeriodTotals[y];

                              //and set the flag in the Counted array
                              Counted[b] = TRUE;
                    }
          }//end for(b=a+1;b<(signed)count;b++)

          //Clear up the TempPeriodTotals[] so it can be used for the next treelist without additive problem
          for(y=0;y<NP;y++)
                    TempPeriodTotals[y] = 0;

}//end for(a=0;a<(signed)count;a++)


//The SUM values currently in SumBigTrees[] need each to be divided by 3 since there were 3 goal scenarios used in
calculating it
for(a=0;a<NP;a++)
          SumBigTrees[a] = SumBigTrees[a] / 3;


//Look through SumBigTrees[] and find the period with the largest value and use that as the return base
for(a=0;a<NP;a++)
{
          if(SumBigTrees[a] > LargestSum)
                    LargestSum = SumBigTrees[a];
}

//Delete stuff on free store
delete [] Counted;

          return LargestSum;
}//end GetSumBigTrees


//*****************************************************************************************************************
***
void SwaplAdjust( struct SOLUTION *ptr_cs, ushort NG, ushort NH, double PerValues[], struct OPTIMIZE_SINGLE_VALUE
OV[],
                              ulong Records, struct ERA *ptr_era)
//*****************************************************************************************************************
***
{
/*
A  move is being tested and needed is to subtract off the treelist-goal-hold "optimizing" values being moved
OUT of the solution and to add the treelist-goal-hold "optimizing" values for that being moved INTO the solution.

ALSO, subtract off the ERA values being moved off and add those values being moved into the Era structure

NOTE:  See long-winded note in CalculateSumPeriodEra() function about theory for this.
```

```
*/

int a,b;


struct OPTIMIZE_SINGLE_VALUE OVKey;
struct OPTIMIZE_SINGLE_VALUE *ptr_ovkey;

struct CURRENT_ERAS CellEraValues, *ptr_cev;

// ------------------------------- End of variable defining ---------------------

//Initialize the OVKey
memset(&OVKey, 0, sizeof(struct OPTIMIZE_SINGLE_VALUE));

//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//++++++++++++++++++++++++++   SUBTRACT OFF VALUES FOR PRESCRIPTION BEING MOVED OUT   ++++++++++++++++++++++++++
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// ========================================================
//                          Make a key for the OV structure
// ========================================================

//First, make a key and look for stand being moved OUT of solution and reduce the PerValues array
OVKey.Treelist      = ptr_cs->Treelist;
OVKey.Goal                  = ptr_cs->Goal;
OVKey.Hold                  = ptr_cs->Hold;


//Now use bsearch to find the matching record in the array of OV structures
ptr_ovkey = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
          &OVKey,
          (void *)OV,
          (size_t)Records,
          sizeof( struct OPTIMIZE_SINGLE_VALUE),
          LookAtOSV );

if(ptr_ovkey == NULL)                                   //There had better be one!
          Bailout(80);

//=================================================================
//        Assume that if to here then everything has been found correctly
//=================================================================

//First, subtract off values from the PerValues[] array
for(a=0;a<NP;a++)
          PerValues[a]                                  -= ptr_ovkey->Value[a];

//Then subtract off the contribution this cell made to the subwatershed ERA value
for(a=0;a<NP;a++)
          ptr_era->SumPeriodEra[a]      -= ptr_cs->PeriodEra[a];

//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//++++++++++++++++++++    CALCULATE NEW VALUES FOR PRESCRIPTION BEING MOVED INTO SOLUTION   ++++++++++++++++++++
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
memset(&OVKey, 0, sizeof(struct OPTIMIZE_SINGLE_VALUE));

//Make a key and find the new prescription values in the OV[] structures
OVKey.Treelist      = ptr_cs->Treelist;
                    //The treelist doesn't change!
OVKey.Goal                  = NG;
OVKey.Hold                  = NH;

//Now use bsearch to find the matching record in the array of OV structures
ptr_ovkey = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
          &OVKey,
          (void *)OV,
          (size_t)Records,
          sizeof( struct OPTIMIZE_SINGLE_VALUE),
          LookAtOSV );

if(ptr_ovkey == NULL)                                   //There had better be one!
          Bailout(80);

//=================================================================
//        Assume that if to here then everything has been found correctly
//=================================================================

//First, ADD the values from the PerValues[] array
for(a=0;a<NP;a++)
          PerValues[a]                              += ptr_ovkey->Value[a];

/*
Figuring out the new ERA values to add is a bit more complicated.  Because I have not precalculated EVERY possible
PeriodEra[] value for every cell and every prescription (which I may want to do), I need to quickly calculate that
value for this new move.  Call the same function as done in CalculateSumPeriodEra() except here I will assume that
the cell is in the solution so I don't need to do all the pre-checking it does.
*/

//clear the CellEraValues stuff before filling and sending off
memset(&CellEraValues, 0, sizeof(struct CURRENT_ERAS) );

//Make a package of stuff to send off to get NetEra's calculated
CellEraValues.ptr_osv                   = ptr_ovkey;
```

```
CellEraValues.CurrentEra      = ( (float)ptr_cs->InitialEra / ERA_EXP ) ;                //last stored as modified
ushort

//Need to send a pointer to get values back
ptr_cev = &CellEraValues;

//Ship pointer off to function which will calculate NetEra's for each period
CalculateNetEras(ptr_cev);


//Store the return values in the NetEra[] member in two places for each period
for(b=0;b<NP;b++)
{
          ptr_cs->PeriodEra[b]                    = (ushort)(ptr_cev->NetEra[b]);
          ptr_era->SumPeriodEra[b]       += (ulong)(ptr_cev->NetEra[b]);
}

}//end SwaplAdjust


//*****************************************************************************************
int DecreaseShort(ulong count, ushort Short[][GOALS][HOLDNO])
//*****************************************************************************************
{
ulong a;
ushort *ptr_short;

//printf("Iterations to do is %lu\n",(signed)count*GOALS*HOLDNO);

#ifdef DEBUG_DECREASESHORT1
int b,c;
//Test, go through Short and see which one has a value, to see if
for(a=0;a<count;a++)
{
          for(b=0;b<GOALS;b++)
          {
                    for(c=0;c<HOLDNO;c++)
                    {
                              if(Short[a][b][c] > 0 )
                                        printf("Move cell %lu, to goal %d with Hold %d\n",a,b,c);
                    }
          }
}
#endif


//Put a pointer at start of Short and go through the entire array and decrease all values
//that are greater than 0 by -1.
ptr_short = &Short[0][0][0];

for(a=0;a<count*GOALS*HOLDNO;a++)
{
          if( *ptr_short > 0)
                    *ptr_short = *ptr_short - 1;
}

          return TRUE;
}//end DecreseShort

//*****************************************************************************************
void InputAndCalculateSolutionEras(ulong Count, struct SOLUTION CS[] )
//*****************************************************************************************
{
/*
CS[] needs to have been sorted by Cellid before entering here.

The goal here is to look at the array of CS structures, which has the Current Solution for the
entire watershed, with 5 member:  Minor - Cellid - Treelist - Goal - HoldFor & PeriodEra[NP].
There are "Count" records of this structure.

This function will go cell-by-cell through the Data.* array and check every cell to see if it
was in the solution.  If so, it will use the values in CS[].PeriodEra[] to fill Data.Era[][].

IF there is no match, then the cell was NOT in the solution, so its initial Data.InitialEra[]
value needs to be packaged and sent to function to get it's proper ERA decay values - then
input those values into Data.Era[][].
*/

int a,b;
int InSolution;

//Key stuff for structures
struct SOLUTION SKey;
struct SOLUTION *ptr_skey;
struct CURRENT_ERAS CellEraValues, *ptr_cev;

//------------------------------- End of variable defining -------------------------------

for(a=0;a<UNIQUE;a++)
{

          if( Data.Cellid[a] == FALSE )                    //no more cells to check
                    break;

          //*****************************************************************
```

```
		//	Determine if cell was actually in the solution
		//*************************************************************
		//Make a key for the current cell using its cellid
		SKey.Cellid			= Data.Cellid[a];

		//Use bsearch on CS[] to see if this cell is in the solution
		ptr_skey  = (struct SOLUTION*)bsearch(
			&SKey,
			(void *)CS,
			(size_t)Count,
			sizeof( struct SOLUTION),
			LookAtSolutionCellid );

		//Make a flag to use below
		if( ptr_skey == NULL )
			InSolution = FALSE; //cell not in solution
		else
			InSolution = TRUE;

		//*************************************************************
		//	THE CELL WAS IN THE SOLUTION
		//*************************************************************
		if( InSolution == TRUE )
		{
			//Just copy over the data stored in the PeriodEra[] member of CS
			for(b=0;b<NP;b++)
				Data.Era[a][b] = ptr_skey->PeriodEra[b];

		}
		else
		{
		//*************************************************************
		//	Cell was NOT in the original solution
		//*************************************************************
		//NOTE: these calculations could probably be done back in CaluculateSumPeriodEra() and stored for these
		//cells not in the solution.

			/*
			Just slowly decay or "recover" cells current Data.InitialEra[] proportionally down to 0.
			There is no documentation to do this but it should not matter because they don't contribute to
anything.
			I am thinking that later we may want to "recover" certain areas at different rates and track
how those
			subwatershed that are "unmanaged" fair compare to those that are managed.
			*/

			//clear the CellEraValues stuff before filling and sending off
			memset(&CellEraValues, 0, sizeof(struct CURRENT_ERAS) );

			//Make a package of stuff to send off to get NetEra's calculated
			CellEraValues.CurrentEra	= ( (float)Data.InitialEra[a] / ERA_EXP ) ;
		//last stored as modified ushort
			CellEraValues.Cell			= a;

			//Need to send a pointer to get values back
			ptr_cev = &CellEraValues;

			//Ship pointer off to function which will calculate DecayOnly NetEra's for each period
			CalculateDecayOnlyNetEras(ptr_cev);

			//If new decayed NetEras were calculated, store their values in Data.Era[][] - otherwise
already initialized to zero
			if( ptr_cev->NeedsDecay == TRUE )
			{
				for(b=0;b<NP;b++)
					Data.Era[a][b] = (ushort)(ptr_cev->NetEra[b]);
			}

		}//end else if(InSolution ...

}//end for(a=0 ... )

}//end InputAndCalculateSolutionEras


//*****************************************************************************
int InputSolution(ulong Count, struct SOLUTION CS[] )
//*****************************************************************************
{
/*
CS[] needs to have been sorted by Cellid before entering here.

The goal here is to look at the array of CS structures, which has the Current Solution for the
entire watershed, with 5 member:  Minor - Cellid - Treelist - Goal - HoldFor.
There are "Count" records of this structure.

I am going to make the assumption that the Cellid's in CS[].Cellid are in ascending (row/column) order, because
they
were sorted by Cellid back in Goal*().  So I will start by looking at the first CS[].Cellid value and
find that CellId number in Data.Cellid (checking to see if CS[].Treelist matches Data.Treelist)...
if all checks out then put the values of CS[].Goal & CS[].Hold into Data.Goal and Data.Hold
*/

ulong *ptr_cellid, *ptr_treelist, Cellid, Treelist;
ushort *ptr_goal, *ptr_hold, Goal, Hold;
```

```
int FoundMatch;
ulong a;

//-------------------------------- End of variable defining --------------------

printf("Inputting the solution just found into Data[].Goal * Data[].Hold\n");

//Put pointers at start of Data.* arrays
ptr_cellid =        &Data.Cellid[0];
ptr_treelist =      &Data.Treelist[0];
ptr_goal =                  &Data.Goal[0];
ptr_hold =                  &Data.Hold[0];



for(a=0;a<Count;a++)
{
          //Get Values for current cell in CS
          Cellid              = CS[a].Cellid;
          Treelist  = CS[a].Treelist;
          Goal                = CS[a].Goal;
          Hold                = CS[a].Hold;

          //Start looking through the Data.* arrays and find a match
          FoundMatch = 0;
          do{
                    if( *ptr_cellid == Cellid )                                   //Ok, the cellid's match, so should
everything else!
                    {

                              if( *ptr_treelist != Treelist)
                                        Bailout(31);
                              else                                                                                  //put
in the Goal and HoldFor values
                              {
                                        *ptr_goal = Goal;
                                        *ptr_hold = Hold;

                                        FoundMatch = 1;
                              }
                    }

                    //increment pointers, whether or not a match was found
                    //REMEMBER- this works because both CS and Data.Cellid have cellid's in "row/column" order
                    ptr_cellid++;
                    ptr_treelist++;
                    ptr_goal++;
                    ptr_hold++;

          }while(FoundMatch == 0);

}//end for(a ...)

          return TRUE;
}//end InputSolution


//*****************************************************************************************
void BinarySaveGoalHold(void)
//*****************************************************************************************
{
/*
This function will spit out the current configuration of Data.Goal and Data.Hold that
was found during the initial landscape optimization.  It will do this by just sending
out all the values, in order, from those arrays.

When reading back in the data there will be no need to check positioning because (I hope)
the data is already in its correct spot. See the bottom of function ReadBinaryFiles() for
how this is done.
*/

FILE *BIN;
char GoalOutFile[150]="";
char HoldOutFile[150]="";

ushort *ptr_goal;
ushort *ptr_hold;

//Make the correct output file names
sprintf(GoalOutFile, "%s%s%d\\%s_%s_goal.bin",PREFIX,RerunDir,GOAL_TO_USE,OPTPREFIX,ENVT);
sprintf(HoldOutFile, "%s%s%d\\%s_%s_hold.bin",PREFIX,RerunDir,GOAL_TO_USE.OPTPREFIX,ENVT);

ptr_goal = &Data.Goal[0];
ptr_hold = &Data.Hold[0];

BIN = fopen(GoalOutFile, "wb");
fwrite(ptr_goal,sizeof(Data.Goal[0]),UNIQUE,BIN);
fclose(BIN);

BIN = fopen(HoldOutFile, "wb");
fwrite(ptr_hold,sizeof(Data.Hold[0]),UNIQUE,BIN);
fclose(BIN);

}//end BinarySaveGoalHold
```

```
//*****************************************************************************
void AsciiSaveGoalHold(void)
//*****************************************************************************
{

printf("\n******* Saving the current configuration of GOALS and HOLD from this simulation *******\n\n");


FILE *WRITE_GOAL, *WRITE_HOLD;
char Temp1[250], Temp2[250];


//pointers
int *ptr_srp;                   //Starting Row Position
ushort  *ptr_goal, *ptr_hold, *ptr_column;

//Misc. variables
int r,c,HowMany;
int ColumnsLeft, ctr;
ushort StartColumn,OutColumn;



//Make the correct output file names
sprintf(Temp1, "%s%s%d\\%s_goal.bin",PREFIX,RerunDir,GOAL_TO_USE,ENVT);
sprintf(Temp2, "%s%s%d\\%s_hold.bin",PREFIX,RerunDir,GOAL_TO_USE,ENVT);



//open up the files (to write)
          WRITE_GOAL= fopen(Temp1, "w");
          WRITE_HOLD = fopen(Temp2, "w");

          if (WRITE_GOAL == NULL)
                    fprintf(stderr, "opening of %s failed: %s\n", Temp1, strerror(errno));
          else
#ifdef DEBUG_OPEN1
                    printf("File %s opened with no problems in mode WRITE!\n",Temp1);
#endif

          if (WRITE_HOLD == NULL)
                    fprintf(stderr, "opening of %s failed: %s\n", Temp2, strerror(errno));
          else
#ifdef DEBUG_OPEN1
                    printf("File %s opened with no problems in mode WRITE!\n",Temp2);
#endif


          //write out the header data to the files
          fprintf(WRITE_GOAL,"ncols\t\t%d\n",COLUMNS);
          fprintf(WRITE_GOAL,"nrows\t\t%d\n",ROWS);
          fprintf(WRITE_GOAL,"xllcorner\t%.6lf\n",F_XLL);
          fprintf(WRITE_GOAL,"yllcorner\t%.6lf\n",F_YLL);
          fprintf(WRITE_GOAL,"cellsize\t%d\n",CELLSIZE);
          fprintf(WRITE_GOAL,"NODATA_value\t%d\n",NODATA);

          fprintf(WRITE_HOLD,"ncols\t\t%d\n",COLUMNS);
          fprintf(WRITE_HOLD,"nrows\t\t%d\n",ROWS);
          fprintf(WRITE_HOLD,"xllcorner\t%.6lf\n",F_XLL);
          fprintf(WRITE_HOLD,"yllcorner\t%.6lf\n",F_YLL);
          fprintf(WRITE_HOLD,"cellsize\t%d\n",CELLSIZE);
          fprintf(WRITE_HOLD,"NODATA_value\t%d\n",NODATA);


                    for(r=1;r<=ROWS;r++)
                    {
                              ptr_srp = &link[r-1][1];
                              HowMany = *(ptr_srp+1);
                              StartColumn = Data.GridColumn[(*ptr_srp)-1];
                              ptr_column = &Data.GridColumn[(*ptr_srp)-1];
                              ptr_goal = &Data.Goal[(*ptr_srp)-1];
                              ptr_hold = &Data.Hold[(*ptr_srp)-1];

                              //If the whole row is blank, print out NODATA and goto next row
                              if( *ptr_srp == FALSE )                 //means a zero was left in this spot during
MakeLink
                              {
                                        for(c=1;c<=COLUMNS;c++)
                                        {
                                                  fprintf(WRITE_GOAL,"%d  ",NODATA);
                                                  fprintf(WRITE_HOLD,"%d  ",NODATA);

                                        }
                                        //put in new lines
                                        fprintf(WRITE_GOAL, "\n");
                                        fprintf(WRITE_HOLD, "\n");

                                        continue;           //goto next row
                              }

                              //print out NODATA for those cells before data starts
                              for(c=1;c<StartColumn;c++)
                              {
                                        fprintf(WRITE_GOAL,"%d  ",NODATA);
```

```
                    fprintf(WRITE_HOLD,"%d  ",NODATA);
                }

                //set some counters
                OutColumn = StartColumn;
                ctr = 0;

                //print out the Goal and Hold values for area on landscape by checking
                //value in Data.GridColumn to match it with OutColumn value
                do{
                    if(*ptr_column == OutColumn)
                    {
                        fprintf(WRITE_GOAL, "%hu  ",*ptr_goal);
                        fprintf(WRITE_HOLD, "%hu  ",*ptr_hold);

                        ptr_goal++;
                        ptr_hold++;
                        ptr_column++;
                        OutColumn++;
                        ctr++;
                    }
                    else    //print out NODATA for the "gaps"
                    {
                        fprintf(WRITE_GOAL,"%d  ",NODATA);
                        fprintf(WRITE_HOLD,"%d  ",NODATA);

                        OutColumn++;
                    }
                }while(ctr != HowMany );

                //Check to see how many columns are left to do
                ColumnsLeft = COLUMNS - (OutColumn-1);

                if(ColumnsLeft == 0)
                {
                    fprintf(WRITE_GOAL, "\n");
                    fprintf(WRITE_HOLD, "\n");

                    continue;                   //go to next row
                }

                //print out NODATA for those cells after the data that are left
                for(c=0;c<ColumnsLeft;c++)
                {
                        fprintf(WRITE_GOAL,"%d  ",NODATA);
                        fprintf(WRITE_HOLD,"%d  ",NODATA);
                }

                //put in a new line
                    fprintf(WRITE_GOAL, "\n");
                    fprintf(WRITE_HOLD, "\n");

            }//end of for(r=1;r<=ROWS;r++)

fclose(WRITE_GOAL);
fclose(WRITE_HOLD);


}//end SaveGoalHold

//*************************************************************************************************************************
*******************
void PrintSolutionValues(ulong Count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[],
                                                    ulong Records, int SubEra[], int Status)
//*************************************************************************************************************************
*******************
{
/*
This functions is to be called up after a heuristic search solution has been completed.
This will use the best solution found to add up the particular value being optimized and print those out.
*/
FILE *WriteOut;
char filename[256];

ulong a;
int b;

double PerValue[NP];
double SumSqDev = 0;
double SumValue = 0;

struct OPTIMIZE_SINGLE_VALUE Key;
struct OPTIMIZE_SINGLE_VALUE *ptr_key;

//-------------------------------------- END Variable defining ------------------------------------------
if(Status == ACTUAL)
        sprintf(filename, "%s%s%d\\Actual_TotalValue.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
else
        sprintf(filename, "%s%s%d\\Reuse_TotalValue.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);



//Open up the file for printing
```

```
WriteOut = fopen(filename, "w");

/*
Using the TREELIST, GOAL, and HOLD in the array of CS[] structures, find the matching set
in the array of OV structures.  Once found, sum up the periodic values and store those
in PerValue[NP],

This functions assumes that the values found in Solution are the current one to evaluate.
*/

//Initialize the PerValue[] array
memset( PerValue, 0, sizeof(PerValue) );


for(a=0;a<Count;a++)          //count is how many rows of data there are (i.e. eligible cells found earlier)
{
        //Make a Key using the Treelist-Goal-Hold values found for each record in the array of CS structures
        Key.Treelist        =  CS[a].Treelist;
        Key.Goal            =  CS[a].Goal;
        Key.Hold            =  CS[a].Hold;


        //Now use bsearch to find the matching record in the array of OV structures
        ptr_key = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
                &Key,
                (void *)OV,
                (size_t)Records,
                sizeof( struct OPTIMIZE_SINGLE_VALUE),
                LookAtOSV );

        if(ptr_key == NULL)                                  //There had better be one!
        {
                printf("Bad Key.Treelist = %lu...CS[].Treelist is %lu\n",Key.Treelist,CS[a].Treelist);
                printf("Bad Key.Goal = %hu...CS[].Goal is %lu\n",Key.Goal,CS[a].Goal);
                printf("Bad Key,Hold = %hu...CS[].Hold is %lu\n",Key.Hold,CS[a].Hold);
                Bailout(80);
        }
        else                                                                    //Sum up the periodic Values
        {
                for(b=0;b<NP;b++)
                                        PerValue[b] += ptr_key->Value[b];
        }
}



// ======================== PRINT OUT STUFF BELOW ===================================

        fprintf(WriteOut,"\nThe Periodic Big Trees Totals are:\n");
        fprintf(WriteOut,"NOTE:  These values are rounded INTEGER BigTrees values and will usually be less than
real value\n\n");

        for(a=0;a<NP;a++)
                fprintf(WriteOut,"Per%d is %-.3lf\n",a+1,PerValue[a]/BIGTREES_EXP);


        fprintf(WriteOut, "\n\nLoopsToDo: %lu\n",Count*LOOP_FACTOR);
        fprintf(WriteOut, "The constraining Sub-Watershed ERA threshold were:\n");
        for(a=0;a<NP;a++)
                fprintf(WriteOut, "Per%d had a ERA threshold of %d\n",a+1, SubEra[a]);

fclose(WriteOut);

}//end PrintSolutionValues


//*****************************************************************************************
void PrintSolutionEraValues(struct ERA Era[], ulong NoSheds, int Status)
//*****************************************************************************************
{
/* This will output a table with 6th field subwatershed id's in Rows, and columns for the
four EvaluateThisPeriod[] periods, with values representing the Equivalent Roaded Acre (ERA)
value for that sub-watershed. This file will be comma delimited and can be imported into ArcInfo and
joined with the SubWatershed layer to make maps showing the ERA's - or the tables can be used stand-alone.

This will only handle the initial ERA values and only ouput 4 periods worth of data.  Another function
will ouput the actual ERA values stored in the Data.*[] arrays after a full simulation.

NOTE: This function is using ERA values only for those cells in a "solution" - which is different than
what gets outputted in OutputEraValues() at end of simulation.
*/

FILE *WRITE_ERA;
char EraFile[256];
int a,b;
int Hit,PerA,PerB,PerC,PerD;
//---------------------------------- End of variable defining --------------------------------

//Look at EvaluateThisPeriod and find the 4 evaluation periods there
Hit=0;
for(a=0;a<NP;a++)
{
        if(EvaluateThisPeriod[a] > 0)
        {
```

```
                    if(Hit == 4)
                              printf("There are too many EvaluateThisPeriod[] periods! - ignoring those past the
first four\n");

                    if(Hit == 0)
                              PerA = a;
                    else if(Hit == 1)
                              PerB = a;
                    else if(Hit == 2)
                              PerC = a;
                    else
                              PerD = a;

                    Hit++;
          }
}//end for(r=0;r<NP;r++)


// Create, Open, and Write data out to a file
if(Status == ACTUAL)
          sprintf(EraFile, "%s%s%d\\Actual_era.csv",PREFIX,PreSimOutputDir,GOAL_TO_USE);
else if(Status == LAST)
          sprintf(EraFile, "%s%s%d\\era.csv",PREFIX,PostSimOutputDir,GOAL_TO_USE);
else
          sprintf(EraFile, "%s%s%d\\Reuse_era.csv",PREFIX,PreSimOutputDir,GOAL_TO_USE);
WRITE_ERA =          fopen(EraFile, "w");
if (WRITE_ERA == NULL)
          fprintf(stderr, "opening of %s failed: %s\n", EraFile, strerror(errno));



//No header line because ArcInfo won't import them
//Will output the actual ERA associated with the solution -- ** no TABS either (A/I doesn't like them)
for(a=0;a<(signed)NoSheds;a++)
{
          fprintf(WRITE_ERA,"%hu,  ",Era[a].Minor);
          for(b=0;b<NP;b++)
          {
                    if(b == PerA || b == PerB || b == PerC || b == PerD)
                    {
                              fprintf(WRITE_ERA,"%.2f  ",( (float)Era[a].SumPeriodEra[b] / ERA_EXP) / Era[a].Count
);

                              if(b != PerD)
                                        fprintf(WRITE_ERA,",");
                    }

          }
          fprintf(WRITE_ERA,"\n");
}


fclose(WRITE_ERA);

}//end PrintSolutionEraValues

//*****************************************************************************************************************
*******************
void PrintSolutionBigTrees(ulong Count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[],
                                                  ulong Records, int SubEra[], int Status)
//*****************************************************************************************************************
*******************
{
/*
This functions is to be called up after a heuristic search solution has been completed.
This will use the best solution found to add up the Big Trees and print those out.

NOTE: This is sorta redundant because for now, goal 1 is optimizing BigTrees and so when the
PrintSolutionValues() gets printed, it should be the same as this - a double check.
*/
FILE *WriteOut;
char filename[256];

ulong a;
int b;

double PerBigTrees[NP], SumBigTrees;

struct OPTIMIZE_SINGLE_VALUE Key;
struct OPTIMIZE_SINGLE_VALUE *ptr_key;

//-------------------------------------- END Variable defining ------------------------------------------
if(Status == ACTUAL)
          sprintf(filename, "%s%s%d\\Actual_BigTrees.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
else
          sprintf(filename, "%s%s%d\\Reuse_BigTrees.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);


//Open up the file for printing
WriteOut = fopen(filename, "w");

/*
Using the TREELIST, GOAL, and HOLD in the array of CS[] structures, find the matching set
in the array of OV structures.  Once found, sum up the BigTrees and store those
```

```
in PerBigTrees[NP],

This functions assumes that the values found in Solution are the current one to evaluate.
*/

//Initialize the PerBigTrees[] array
memset( PerBigTrees, 0, sizeof(PerBigTrees) );


for(a=0;a<Count;a++)          //count is how many rows of data there are (i.e. eligible cells found earlier)
{
          //Make a Key using the Treelist-Goal-Hold values found for each record in the array of CS structures
          Key.Treelist        =  CS[a].Treelist;
          Key.Goal            =  CS[a].Goal;
          Key.Hold            =  CS[a].Hold;


          //Now use bsearch to find the matching record in the array of OV structures
          ptr_key = (struct OPTIMIZE_SINGLE_VALUE*)bsearch(
                    &Key,
                    (void *)OV,
                    (size_t)Records,
                    sizeof( struct OPTIMIZE_SINGLE_VALUE),
                    LookAtOSV );

          if(ptr_key == NULL)                              //There had better be one!
          {
                    printf("Bad Key.Treelist = %lu...CS[].Treelist is %lu\n",Key.Treelist,CS[a].Treelist);
                    printf("Bad Key.Goal = %hu...CS[].Goal is %lu\n",Key.Goal,CS[a].Goal);
                    printf("Bad Key,Hold = %hu...CS[].Hold is %lu\n",Key.Hold,CS[a].Hold);
                    Bailout(80);
          }
          else                                                          //Sum up the periodic Values
          {
                    for(b=0;b<NP;b++)
                              PerBigTrees[b] += ptr_key->BigTrees[b];
          }
}


//Add up the total sum of big trees
for(b=0;b<NP;b++)
          SumBigTrees += PerBigTrees[b]/BIGTREES_EXP;

// ======================== PRINT OUT STUFF BELOW ==================================

          fprintf(WriteOut,"\nThe Periodic Big Trees Totals are:\n");
          for(a=0;a<NP;a++)
                    fprintf(WriteOut,"Per%d is %-.3lf\n",a+1,PerBigTrees[a]/BIGTREES_EXP);

          fprintf(WriteOut,"\n\nThe total sum of Big Trees is:  %.3lf\n",SumBigTrees);
          fprintf(WriteOut,"Which amounts to about %.3lf per acre\n",SumBigTrees/(Count*ACREEQ));

fclose(WriteOut);

}//end PrintSolutionBigTrees


GOAL_REUSE.CPP

/*
*****************************************************************************************
This file will contain functions to control how to reuse data for prescriptions that
were already selected.
*****************************************************************************************
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"
#include "goals.h"            //for functions declared in goal_controller.cpp

//Functions used in this file
void ReuseGoal(int Goal);

//Declared in misc.cpp
extern int FillSubEraThresholds(int SubEra[]);

//Declared in ReadData.cpp
extern void ReadGoalHoldFound(int Goal);

//*****************************************************************************
void ReuseGoal(int Goal)
//*****************************************************************************
{
/*
This function will assume that a GOAL-HOLD combination was already found for the particular
landscape being used and that binary files with the goal & hold are in the
...\RerunData\ directory.  Because many of the output analysis data functions need to know
what the "solution space" (i.e. those cells that were actually in the solution) was, I found
it easier to pretend that an optimization process is happening except skip the heuristic part
```

```
since there is already an answer.
*/

FILE *BinIn, *HeaderIn;
char Temp[256];
ulong Records;

int a;
ulong AllocOK, AllocNOK,CellsInShed;
ulong SolutionCounters[3];
ulong SolutionSheds;
int SubEra[NP];

//---------------------------------- End of variable defining ----------------------------
printf("\n\n************************* Reusing and Recreating a solution for this goal %d
**********************\n",Goal);

//First thing is to read in the GOAL & HOLD values already found
ReadGoalHoldFound(Goal);

//Fill the FillValueToOptimize array before starting
if( FillValueToOptimize() == FALSE)
          Bailout(24);

//================================= READ in the InitOpt.bin file ================================================
//Open the Header and actual Binary file containing the data found during FillValueToOptimize()
sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.bin",PREFIX,InitialStandDataDir,ENVT);
BinIn = fopen(Temp, "rb");

sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderIn = fopen(Temp, "r");

//Get the Number of records that are listed in the header file
fscanf(HeaderIn,"%lu",&Records);

//Create an array of structures on the free store to hold these records
struct OPTIMIZE_SINGLE_VALUE (*OptValues) = new struct OPTIMIZE_SINGLE_VALUE[Records];
if( OptValues == NULL )
          printf("Problems allocating memory for OptValues[] with %lu
elements\n",Records*sizeof(OPTIMIZE_SINGLE_VALUE));


//Now just read in the binary data the same way it was written out in FillValueToOptimize()
fread(OptValues, sizeof(OPTIMIZE_SINGLE_VALUE),Records,BinIn);

//close up the files
fclose(BinIn);
fclose(HeaderIn);
//=================================== Finished reading
================================================================

//Initialize the SolutionCounters array and call up the DetermineEligibleCells() function to fill it up
for(a=0;a<3;a++)
          SolutionCounters[a] = 0;

printf("*** Going to determine the eligible cells for this solution and fill up the array of SOLUTION structures
***\n");

if( DetermineEligibleCells(SolutionCounters) == FALSE)
          Bailout(82);


//The values now in SolutionCounters should be properly set
AllocOK             = SolutionCounters[0];
AllocNOK  = SolutionCounters[1];
CellsInShed         = SolutionCounters[2];

printf("!!! There were %lu valid cells with cellids.....",CellsInShed);
printf(" and %lu cells that are eligible for the solution and %lu that are not.\n\n",AllocOK,AllocNOK);

//Print stuff to the stats.txt file
PrintToStat(3, (ulong)AllocOK);

//Set a checker to look for when there are 0 eligible cells
if(AllocOK == FALSE)
          Bailout(89);

//Create an array of structures on the free store to hold the solution
struct SOLUTION (*CurrentSolution) = new struct SOLUTION[AllocOK];
if( CurrentSolution == NULL )
          printf("Problems allocating memory for CurrentSolution[] with %lu elements\n",AllocOK*sizeof(SOLUTION));

//Initialize
memset( CurrentSolution, 0, sizeof(struct SOLUTION) * AllocOK );

//Now fill that array of SOLUTION structures with the Treelist - Minor - and Cellid of those eligible cells
if( FillSolution(SolutionCounters, CurrentSolution, FAKE) == FALSE )
          Bailout(83);


printf("Sorting the solution by subwatersheds....will take a few seconds\n");
//Now sort the array of SOLUTION structures by MINOR .  This will guarantee all the subwatersheds are in order
//Use mgsort because qsort takes way too long since there are not many unique Minor ID's
mgsort(   (void*)CurrentSolution,                                   //base
                    (size_t)AllocOK,                                          //count of # of arrays
```

```
                    sizeof(struct SOLUTION),                    //size of each array
                    0, AllocOK-1,                                       //current division (
always: 0, "Count"-1 )
                    LookAtSolutionMinor );                              //compare function


//Call up the CountSolutionWatersheds() function to see how many subwatersheds are actually in the solution
SolutionSheds = CountSolutionWatersheds(AllocOK, CurrentSolution);
printf("*** There were actually %lu Sub-Watersheds found in the solution for this goal ***\n",SolutionSheds);

if( SolutionSheds == FALSE )
          Bailout(84);

//Create the appropriate number of Solution_ERA structures and store them in an array
struct ERA (*S_Era) = new struct ERA[SolutionSheds];
if(S_Era == NULL)
          printf("Problems allocating memory for S_Era with %lu elements\n",SolutionSheds*sizeof(struct ERA));

//Initialize this array of ERA structures - this is important because Fill_SEra will do some += summing
memset( S_Era, 0, sizeof(struct ERA) * SolutionSheds );


//Fill the array of S_Era structures with appropriate values
if( Fill_SEra(SolutionSheds, S_Era, AllocOK, CurrentSolution) == FALSE )
          Bailout(85);

//First, fill up the SubEra array with ERA thresholds to use when checking constraints
if( FillSubEraThresholds(SubEra) == FALSE )
          Bailout(94);

//First sort the  array of CS structures by CELLID only. This is needed later
qsort(    (void*)CurrentSolution,                                   //base
                    (size_t)AllocOK,                                        //count of # of
arrays
                    sizeof(struct SOLUTION),                          //size of each array
                    LookAtSolutionCellid          );                        //compare
function


if( CalculateSumPeriodEra(SolutionSheds, S_Era, AllocOK, CurrentSolution, OptValues, Records) == FALSE)
          Bailout(86);


//*****************************************************************************************************************
**************
//****************** Stuff below would normally be found in a goal*_"heuristic".cpp file (e.g. goal1_deluge.cpp)
******************
//*****************************************************************************************************************
**************

          //************************ These will print out ONLY those things for cells IN THE SOLUTION
*****************

if( Goal != GROW_ONLY || Goal != FINNEY_EFFECT )  //Grow-only & Finney-Effect didn't have any "value"
{
          //Print out the Periodic values for the Value being optimized
          PrintSolutionValues(AllocOK, CurrentSolution, OptValues, Records, SubEra,REUSE);
}

//Print out the Big Trees
PrintSolutionBigTrees(AllocOK, CurrentSolution, OptValues, Records, SubEra, REUSE);

//Print out the ERA values in S_Era
PrintSolutionEraValues(S_Era,SolutionSheds, REUSE);


//*****************************************************************************************************************
**************
//****************** Stuff below would normally be found at end of goal*.cpp - after the heuristic code file is done
******************
//*****************************************************************************************************************
**************

//NOTE: no need to call the InputSolution() function because it only fills up the Data.Goal and Hold arrays
// and they already are (back in ReadGoalHoldFound() )

//Input the associated PeriodEra[] values found for those cells in the solution, and calculate
//new Data.Era[][] values for those cells that were not in the solution.
InputAndCalculateSolutionEras(AllocOK, CurrentSolution);

//Delete stuff on Free store
delete [] OptValues;
delete [] CurrentSolution;
delete [] S_Era;

}//end ReuseGoal
```

GOAL1.CPP

```
/*
*******************************************************************************************
This file will acts as a subordinate controller (to goal_controller.cpp) - specifically for Goal1.
It can call up either a TabuSearch or Deluge search for this particular goal.  It does rely on some
functions in goal_controller.cpp.
*******************************************************************************************
```

```
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"
#include "goals.h"            //for functions declared in goal_controller.cpp

//Functions used in this file
void Goall(void);
int InitialSolutionGoall(ulong count, struct SOLUTION CS[]);


//declared in goall_deluge.cpp
extern int DelugeGoall(ulong count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[], ulong Records,

                    struct ERA S_Era[], ulong SolutionSheds);

//declared in Constraints.cpp
extern int CheckConstraintsGoall(struct ERA Era[], ulong NoSheds, int SubEra[]);

//Declared in misc.cpp
extern int FillSubEraThresholds(int SubEra[]);


//*****************************************************************************
void Goall(void)
//*****************************************************************************
{
FILE *BinIn, *HeaderIn;
char Temp[256];
ulong Records;

int a, b;
ulong AllocOK, AllocNOK,CellsInShed;
ulong SolutionCounters[3];                              //will get filled with  AllocOK,  AllocNOK,
CellsInShed, by DetermineEligibleCells()
ulong SolutionSheds;
int SubEra[NP];
//------------------------------------ End variable defining --------------------------------------------------

//Fill the FillValueToOptimize array before starting
if( FillValueToOptimize() == FALSE)
         Bailout(24);

//================================= READ in the InitOpt.bin file ================================================
//Open the Header and actual Binary file containing the data found during FillValueToOptimize()
sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.bin",PREFIX,InitialStandDataDir,ENVT);
BinIn = fopen(Temp, "rb");

sprintf(Temp, "%s%s\\Binary\\%s_InitOpt.hdr",PREFIX,InitialStandDataDir,ENVT);
HeaderIn = fopen(Temp, "r");

//Get the Number of records that are listed in the header file
fscanf(HeaderIn,"%lu",&Records);

//Create an array of structures on the free store to hold these records
struct OPTIMIZE_SINGLE_VALUE (*OptValues) = new struct OPTIMIZE_SINGLE_VALUE[Records];
if( OptValues == NULL )
         printf("Problems allocating memory for OptValues[] with %lu
elements\n",Records*sizeof(OPTIMIZE_SINGLE_VALUE));


//Now just read in the binary data the same way it was written out in FillValueToOptimize()
fread(OptValues, sizeof(OPTIMIZE_SINGLE_VALUE),Records,BinIn);

//close up the files
fclose(BinIn);
fclose(HeaderIn);
//=================================== Finished reading
==============================================================

//Initialize the SolutionCounters array and call up the DetermineEligibleCells() function to fill it up
for(a=0;a<3;a++)
         SolutionCounters[a] = 0;

printf("**** Going to determine the eligible cells for this solution and fill up the array of SOLUTION structures
***\n");

if( DetermineEligibleCells(SolutionCounters) == FALSE)
         Bailout(82);


//The values now in SolutionCounters should be properly set
AllocOK             = SolutionCounters[0];
AllocNOK  = SolutionCounters[1];
CellsInShed         = SolutionCounters[2];

printf("!!! There are %lu valid cells with cellids.....",CellsInShed);
printf(" and %lu cells that are eligible for the solution and %lu that are not.\n\n",AllocOK,AllocNOK);

//Print stuff to the stats.txt file
```

```
PrintToStat(3, (ulong)AllocOK);

//Set a checker to look for when there are 0 eligible cells
if(AllocOK == FALSE)
        Bailout(89);

//Create an array of structures on the free store to hold the solution
struct SOLUTION (*CurrentSolution) = new struct SOLUTION[AllocOK];
if( CurrentSolution == NULL )
        printf("Problems allocating memory for CurrentSolution[] with %lu elements\n",AllocOK*sizeof(SOLUTION));

//Initialize
memset( CurrentSolution, 0, sizeof(struct SOLUTION) * AllocOK );

//Now fill that array of SOLUTION structures with the Treelist - Minor - and Cellid of those eligible cells
if( FillSolution(SolutionCounters, CurrentSolution, REAL) == FALSE )
        Bailout(83);


printf("Sorting the solution by subwatersheds....will take a few seconds\n");
//Now sort the array of SOLUTION structures by MINOR .  This will guarantee all the subwatersheds are in order
//Use mgsort because qsort takes way too long since there are not many unique Minor ID's
mgsort(   (void*)CurrentSolution,                                   //base
                (size_t)AllocOK,                                            //count of # of arrays
                sizeof(struct SOLUTION),                            //size of each array
                0, AllocOK-1,                                               //current division (
always: 0, "Count"-1 )
                LookAtSolutionMinor );                                      //compare function


//Call up the CountSolutionWatersheds() function to see how many subwatersheds are actually in the solution
SolutionSheds = CountSolutionWatersheds(AllocOK, CurrentSolution);
printf("*** There were actually %lu Sub-Watersheds found in the solution for this goal ***\n",SolutionSheds);

if( SolutionSheds == FALSE )
        Bailout(84);

//Create the appropriate number of Solution_ERA structures and store them in an array
struct ERA (*S_Era) = new struct ERA[SolutionSheds];
if(S_Era == NULL)
        printf("Problems allocating memory for S_Era with %lu elements\n",SolutionSheds*sizeof(struct ERA));

//Initialize this array of ERA structures - this is important because Fill_SEra will do some += summing
memset( S_Era, 0, sizeof(struct ERA) * SolutionSheds );


//Fill the array of S_Era structures with appropriate values
if( Fill_SEra(SolutionSheds, S_Era, AllocOK, CurrentSolution) == FALSE )
        Bailout(85);

//First, fill up the SubEra array with ERA thresholds to use when checking constraints
if( FillSubEraThresholds(SubEra) == FALSE )
        Bailout(94);

//First sort the  array of CS structures by CELLID only. This is needed later
qsort(    (void*)CurrentSolution,                                                   //base
                (size_t)AllocOK,                                                            //count of # of
arrays
                sizeof(struct SOLUTION),                                            //size of each array
                LookAtSolutionCellid        );                                              //compare
function


//=============================================================================================================
//                                      GET AN INITIAL SOLUTION
//=============================================================================================================
printf("*** Starting to look for an initial solution for this goal that meets all the constraints ***\n");

//Set an error checker for this initial solution - if it fails X times then bailout
for(b=0;b<INITIAL_TRYS;b++)
{
        //Send the Solution to a function to get a random initial solution
        if( InitialSolutionGoal1(AllocOK, CurrentSolution) != TRUE)
                Bailout(25);

        if( CalculateSumPeriodEra(SolutionSheds, S_Era, AllocOK, CurrentSolution, OptValues, Records) == FALSE )
                Bailout(86);

        if( CheckConstraintsGoal1(S_Era,SolutionSheds, SubEra) == FALSE)
                printf("!!!!!!!!! This initial solution failed - trying another !!!!!!!!!\n");
        else
                break;

        if(b == INITIAL_TRYS-1)                 //if it gets to here then X solutions failed so bailout
                Bailout(87);
}

//=============================================================================================================
//                                      Print out the initial solution for later evaluation
//=============================================================================================================
printf("Inputting and printing out the initial goal assignment to the ...\\presimdata\\goal%d
directory\n",GOAL_TO_USE);

//Start by inputting the random solution into the Data.* arrays and pretending it was the final solution
if( InputSolution(AllocOK, CurrentSolution) == FALSE )
```

```
        Bailout(27);

//Now print that file out
OutputInitialGoal();

//=============================================================================================================
//                                                  CALLING UP THE HEURISTIC
//=============================================================================================================
//OK, lets send everything to the appropriate HEURISTIC search function
if( DelugeGoal1(AllocOK, CurrentSolution, OptValues, Records, S_Era, SolutionSheds) != TRUE)
        Bailout(44);
//=============================================================================================================
//                                                  finished with heuristic
//=============================================================================================================


//Input the current solution into the Data.Goal and Data.Hold arrays - regardless if Tabu or Deluge did it!
if( InputSolution(AllocOK, CurrentSolution) == FALSE )
        Bailout(27);

//Also input the associated PeriodEra[] values found for those cells in the solution, and calculate
//new Data.Era[][] values for those cells that were not in the solution.
InputAndCalculateSolutionEras(AllocOK, CurrentSolution);


//Delete stuff on the free store
delete [] S_Era;
delete [] OptValues;
delete [] CurrentSolution;


}       //end of Goall()


//*************************************************************************************************************
int InitialSolutionGoal1(ulong count, struct SOLUTION CS[] )
//*************************************************************************************************************
{
/* The object here to to just assign a random solution to the array of CS[] structures.

The GOAL number must be 0 - 9 inclusive.  See StandOptStuff.cpp for what the goal numbers mean.
The HOLDFOR must be 0 or 3.
*/

int a;
int RGoal, RHold;

//---------------------------- End of variable defining ----------------------------------
printf("Generating random Goal and Hold values\n");

for(a=0;a<(signed)count;a++)
{
        //Get the random Goal and Hold values
        RGoal = ( rand() % 10);                         //this give 0-9

        do{
                RHold = ( rand() % 4);                  //this gives 0,1,2,and 3....but it has to be either 0
or 3
        }while(RHold == 1 || RHold == 2);

        //Store in CS[]
        CS[a].Goal          = (ushort)RGoal;
        CS[a].Hold          = (ushort)RHold;

}//end (a=0;a<count;a++)

        return TRUE;
}//end of InitialSolutionGoal1
```

GOAL1_DELUGE.CPP

```
/*
*************************************************************************************************
This file will hold the specific functions used for a Great Deluge Search on Goal1.
*************************************************************************************************
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"
#include "goals.h"            //for functions declared in goal_controller.cpp


//Declared in this file
int DelugeGoal1(ulong count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[], ulong Records,

                                        struct ERA S_Era[], ulong SolutionSheds);
//Declared in Constraints.cpp
extern int CheckConstraintsGoal1(struct ERA Era[], ulong NoSheds, int SubEra[]);

//Declared in misc.cpp
```

```
extern int FillSubEraThresholds(int SubEra[]);


//-------------------------------- End of function declarations ----------------------------------------
//*****************************************************************************************
int DelugeGoall(ulong count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[], ulong Records,

                                    struct ERA S_Era[], ulong SolutionSheds)

//*************************************************************************************************
{

FILE *WRITE_BEST, *WRITE_RANDOM;
char filename_best[256], Temp[256];

//For Time information
double Start, Finish;
double Duration;


double PerValues[NP], PreMoveValues[NP];
double InitialObj=0;
double LEVEL, PreMoveObj, MoveObj, BestObj,TestObj[GOALS], BestTestObj;
ulong LoopsToDo=0, FirstThird, SecondThird;
ushort PreMovePeriodEra[NP];
ushort *ptr_cellera;

int a,b,x,loop, CopyToPre, ViolateConstraints, BetterThanZero;
float RandCore;

//stuff for the single swap move
ushort PreMoveGoal, PreMoveHold, MoveGoal, MoveHold, BestTestGoal;
ulong RCell;

//Key and pointers for structure stuff
struct SOLUTION *ptr_cs;
struct ERA EKey;
struct ERA *ptr_ekey;

//Array to hold the varying Subwatershed ERA values
int SubEra[NP];
//------------------------------------------- End of variable defining ---------------------------------------
-

puts("\t\t*****************************************************************");
puts("\t\t******                                                    *******");
printf("\t\t******   Starting GREAT DELUGE for GOAL #%d\t\t******\n",GOAL_TO_USE);
puts("\t\t******                                                    *******");
puts("\t\t*****************************************************************");

Start = clock();

//Seed the random number generator
srand(time(NULL));


printf("NOTE: There are %lu cells in the solution for this goal\n",count);


//First, again fill up the SubEra array with ERA thresholds to use when checking constraints -
if( FillSubEraThresholds(SubEra) == FALSE )
        Bailout(94);


//Also create an array of structures to hold a PreMove copy of the current S_Era[] & Copy of the best S_Era found
struct ERA (*PreMoveEra) = new struct ERA[SolutionSheds];
struct ERA (*BestEra)          = new struct ERA[SolutionSheds];
if(PreMoveEra == NULL)
        printf("Problems allocating memory for PreMoveEra with %lu elements\n",SolutionSheds*sizeof(struct ERA));
if(BestEra == NULL)
        printf("Problems allocating memory for BestEra with %lu elements\n",SolutionSheds*sizeof(struct ERA));
//Initialize PreMoveEra & BestEra
memset( PreMoveEra, 0, sizeof(struct ERA) * SolutionSheds );
memset( BestEra, 0 , sizeof(struct ERA) * SolutionSheds);

//Also initialize the PerValues and PreMoveValues and PreMovePeriodEra[] array
memset( PerValues, 0, sizeof(PerValues) );
memset( PreMoveValues, 0, sizeof(PreMoveValues) );
memset( PreMovePeriodEra, 0, sizeof(PreMovePeriodEra) );

/*
Fill PerValues[] with the appropriate values.
PerValues[] is KEY! It has the total "optimizing value" for the Pre-Move solution, for each of the periods.
This array can then be modified - the "value" for the unit-period being moved OUT of will be reduced
and the unit-period being moved INTO will be increased.  After evaluation, PreMoveValues[] will be reinserted

NOTE:  S_Era[] works the same way except it was initially filled up back in Fill_SEra() & CalculateSumPeriodEra()
---which were called while trying to establish the initial solution.
*/
if( Fill_PValues(count, CS, Records, OV, PerValues) == FALSE )
        Bailout(86);

//Create an array of structures on the free store to hold a copy of the BEST solution
struct SOLUTION (*Best) = new struct SOLUTION[count];
if( Best == NULL )
```

```
          printf("Problems allocating memory for Best[] with %lu elements\n",count*sizeof(struct SOLUTION));

//Initialize and copy the initial solution found into this array of Best[] & PreMoveCS[] structures
memset( Best, 0, sizeof(struct SOLUTION) * count );
memcpy( Best, CS, sizeof(struct SOLUTION) * count);

//Also copy the current S_Era into the BestEra
memcpy( BestEra, S_Era, sizeof(struct ERA) * SolutionSheds);

//NOTE: 11Feb00 - skipping this baseline stuff unless needed at later time
          //First, send off the InitialSolution to get a Baseline; Do we really need to get a baseline?
          //Baseline = GetBaselineVTO(count, CS, PerValues, OV, Records);
          //Multiply the Baseline by NP because the objective will be to Maximize the Total # of Big Trees -
regardless of when
          //Baseline = Baseline * NP;


//Get the Initial Objective Value       - just sum up all the trees in PerValues
for(b=0;b<NP;b++)
          InitialObj += PerValues[b] ;


//Set some other objective value holders that will change
LEVEL = InitialObj;
PreMoveObj = InitialObj;                                                    //Make sure to reset after making a new
move!!
BestObj = InitialObj;                                                       //reset as new best are found

printf("InitialObj is %.3lf\n",InitialObj);
printf("LEVEL is %.3lf\n",LEVEL);

//Create, open,and write out the Initial Objective Value to a file for tracking all the best moves
sprintf(filename_best, "%s%s%d\\best.txt",PREFIX,OutputDelugeDir,GOAL_TO_USE);
WRITE_BEST = fopen(filename_best, "w");          //I'm not doing any error-checking here!
fprintf(WRITE_BEST, "%.4lf\t%.4lf\n",InitialObj,LEVEL);

//Also create and open the file to show which cells and goals are being changed - to evaluate if randomness is
working in heuristic
sprintf(Temp, "%s%s%d\\random.txt",PREFIX,PreSimOutputDir,GOAL_TO_USE);
WRITE_RANDOM = fopen(Temp, "w");
if( WRITE_RANDOM == NULL )
          printf("Something wrong opening the %s file\n",Temp);


//*****************************************************************************************************************
******************
//                                                                                                          Start the
Deluge Loop
//*****************************************************************************************************************
******************
//Make the looping a function of how many cells are actually in the current solution
LoopsToDo = (ulong)(count * LOOP_FACTOR);                                   //count was passed in and represents # of
cells in the solution
FirstThird = (ulong)(LoopsToDo * .333333);
SecondThird = (ulong)FirstThird * 2;

printf("Going to do %lu loops\n",LoopsToDo);

CopyToPre = TRUE;
for(loop=0;loop<(signed)LoopsToDo;loop++)
{

          //Always zero out the MoveObj
          MoveObj = 0;

          //Copy the PerValues &the S_Era structures - unless a previous move failed, then they are already set
          if(CopyToPre == TRUE)
          {
                    memcpy(PreMoveValues, PerValues, sizeof(PerValues));
                    memcpy(PreMoveEra, S_Era, sizeof(struct ERA)*SolutionSheds );
          }

          //Pick a random cell in the CS array to move
          RandCore = (float)(rand() / (float)RAND_MAX);
          RCell = (ulong)(RandCore * (count-1));            //will get 0 to "count" [to use in array notation]

          PreMoveGoal         = CS[RCell].Goal;
          PreMoveHold = CS[RCell].Hold;

          //Set a pointer for this cell in the array of CS structures
          ptr_cs = &CS[RCell];

          //Also set a pointer to the current cells values in PeriodEra[]
          ptr_cellera = &CS[RCell].PeriodEra[0];

          //Make a copy of the cells PeriodEra[] member to copy back if final move is no good
          memcpy(PreMovePeriodEra, ptr_cellera, sizeof(PreMovePeriodEra) );

          //a quick bailout if HOLDNO is not correct
          if(HOLDNO > 2)
                    Bailout(28);

          //Pick a new hold, but DO allow it to be same as PreMoveHold - can change later if want to exclude
          do{
                    MoveHold = (ushort)( rand() % 4 );
```

```
        }while(MoveHold == 1 || MoveHold == 2);

        //Make a pointer to the proper S_Era record - to pass to the SwaplAdjust function
        memset(&EKey, 0 , sizeof(struct ERA));

        EKey.Minor          = ptr_cs->Minor;

        //Now use bsearch to find the matching Subwatershed in the array of Era structures
        ptr_ekey = (struct ERA*)bsearch(
                &EKey,
                (void *)S_Era,
                (size_t)SolutionSheds,
                sizeof( struct ERA),
                CompareEraMinor );

        if(ptr_ekey == NULL)                                    //There had better be one!
                Bailout(88);

        /*****************************************************************************************************
*
        Everything above gets done only once per new move (i.e. picking a new cell to change).  What happens next
        is that I will check a small "neighborhood" by evaluating all the stand goals - storing their obj. value
        and then picking the stand goal that made the best move (also checking constraints).
        *****************************************************************************************************
*/
        //Always reinitialize TestObj before next testing loop
        memset( TestObj, 0 , sizeof(TestObj) );

        for(a=0;a<GOALS;a++)
        {
                //reset some variables
                ViolateConstraints = FALSE;

                //Don't evaluate the current stand goal assignment for this cell - has 0 in TestObj[] already
                if( a == PreMoveGoal )
                        continue;

                //Otherwise, call up SwaplAdjust with the current stand goal
                //Note:  CS.Goal & CS.Hold = not changed,
                //BUT, CS.PeriodEra[], PerValues[], & S_Era.SumPeriodEra[] HAVE been changed!
                SwaplAdjust(ptr_cs, a, MoveHold, PerValues, OV, Records, ptr_ekey);


                //Check to see if this test move violates constraints
                if( CheckConstraintsGoall(S_Era,SolutionSheds, SubEra) == FALSE)
                        ViolateConstraints = TRUE;

                //Calculate the TestObj value if constraints not violated - if constraints violated, TestObj[]
has 0 already
                if( ViolateConstraints == FALSE )
                {
                        for(x=0;x<NP;x++)
                                        TestObj[a] += PerValues[x] ;
                }

                //No matter whether or not this test move gets picked, reset some values that got adjusted
during SwaplAdjust
                //These always reset to those values found when first picking which cell to move
                memcpy(PerValues, PreMoveValues, sizeof(PreMoveValues) );
                memcpy(S_Era, PreMoveEra, sizeof(struct ERA)*SolutionSheds );
                memcpy(ptr_cellera, PreMovePeriodEra, sizeof(PreMovePeriodEra) );

        }//end for(a=0 ...
        //*****************************************************************************************************
**********
        //                  End of evaluating neighborhood for this particular cell
        //*****************************************************************************************************
**********

        //Now look through the TestObj[] array and find which stand goal made the largest value - if they are all
the same
        //(and/or all equal to zero) then don't make any move with this cell and skip rest of loop and pick
another cell.

        BestTestObj = 0;
        BetterThanZero = FALSE;
        for(a=0;a<GOALS;a++)
        {
                if(TestObj[a] > BestTestObj )
                {
                        BetterThanZero = TRUE;
                        BestTestGoal = (signed)a;
                        BestTestObj = TestObj[a];
                }
        }//end for(a=0 ...


        //If nothing BetterThanZero was found, then all neighborhood moves violated the constraints - skip rest
and pick another cell
        if( BetterThanZero == FALSE )
        {
                CopyToPre = FALSE;
                continue;
        }
        else
```

```
                CopyToPre = TRUE;

        //++++++++++++++++++++++  IF TO HERE THEN A MOVE WAS MADE AND PASSED CONSTRAINTS ++++++++++++++++++++

        /*
        If to here, then at least one stand goal got a valid answer (although not necessarily a better one - that
        will be checked below).  Remember, that the current stand goal assignment should not have been picked
because it had
        a TempObj[] value of 0, but double-check.
        NOTE:  constraints were already checked - if it failed then it received a zero in TestObj[] and should
not be picked anyways!
        */

        //double checked that current stand goal was not picked
        if( BestTestGoal == PreMoveGoal )
                Bailout(105);

        //Switch BestTestGoal value back to original variable called MoveGoal
        MoveGoal = BestTestGoal;

        //Call up SwaplAdjust again
        //Remember everything was reset back to original values after looking at all the stand goals (the
neighborhood analysis)
        SwaplAdjust(ptr_cs, MoveGoal, MoveHold, PerValues, OV, Records, ptr_ekey);


        //Print out random move information to the random.txt file
        if(loop % PRINT_LOOPS == 0)             //This will be printed every "X" loops
                fprintf(WRITE_RANDOM, "%lu          \t%hu\t%hu
\t%hu\t%hu\n",RCell,PreMoveGoal,PreMoveHold,MoveGoal,MoveHold);

        //See what the MoveObj is
        for(x=0;x<NP;x++)
                        MoveObj += PerValues[x];

        //double-checked that this MoveObj is the same as was calculated earlier
        if( MoveObj != BestTestObj )
                Bailout(106);

        if(loop % PRINT_LOOPS == 0)             //This will be printed first, every "X" loops
        {
                printf("\nJust had a MoveObj of %.4lf and...", MoveObj);
                fprintf(WRITE_BEST, "At loop %d, the current BEST is %.2lf\n",loop, BestObj);
        }

//=================================  EVALUATION PROCESS  ==========================================================

        //This is a MAXIMIZATION problem - if MoveObj is > BestObj then it is already better, so just accept and
go to next loop
        if(MoveObj > BestObj)
        {
                if(loop % PRINT_LOOPS == 0)
                        printf("Was a BEST move,  LEVEL is %.4lf, Best WAS %.2lf, at loop
%d",LEVEL,BestObj,loop);

                //if this is the case, then current PerValues[] & S_Era are OK, but CS[] needs updated -
CS[].PeriodEra[] also OK
                CS[RCell].Goal = MoveGoal;
                CS[RCell].Hold = MoveHold;

                //Reset BestObj
                BestObj = MoveObj;

                //printf("Raising LEVEL from %.6lf to %.6lf\n",LEVEL,LEVEL+RAIN);
                LEVEL += RAIN;

                //write out to file
                //fprintf(WRITE_BEST, "%.4lf\t%.4lf\n",MoveObj,LEVEL);

                //Save CS[] & S_Era[] separately so I can print out later
                memcpy( Best, CS, sizeof(struct SOLUTION) * count);
                memcpy( BestEra, S_Era, sizeof(struct ERA) * SolutionSheds);

        }
        else                                    //an INFERIOR move - use the GREAT DELUGE to decide if still want
to make
        {
                //======== Use the Great Deluge logic ======================
                if(MoveObj > LEVEL)                                             //Yes, I do want to keep
it
                {
                        if(loop % PRINT_LOOPS == 0 )
                                printf("Using GreatDeluge...KEEPING, LEVEL at %.2lf, BEST at %.2lf, at loop
%d",LEVEL,BestObj,loop);

                        //if this is the case, then PerValues[] & S_Era are OK, but CS[] needs updated -
CS[].PeriodEra[] also OK
                        CS[RCell].Goal = MoveGoal;
                        CS[RCell].Hold = MoveHold;


                        //Adjust Level (raise it - maximization problem!)
                        LEVEL += RAIN;
                }
```

```
                else                                                                    //NO, reject the inferior
solution...reset PerValues[] & S_Era, but CS[] is OK
                {
                        if(loop % PRINT_LOOPS == 0 )
                                printf("Using GreatDeluge...REJECTING, LEVEL at %.2lf, BEST at %.2lf, at
loop %d",LEVEL, BestObj,loop);

                        if(loop > (signed)(LoopsToDo - 1000000) )
                        {
                                if(LEVEL >= BestObj)
                                        LEVEL = LEVEL - 50;
                        }

                        //Move has been rejected - reset everything back to PreMove* values
                        memcpy(PerValues, PreMoveValues, sizeof(PreMoveValues) );
                        memcpy(S_Era, PreMoveEra, sizeof(struct ERA)*SolutionSheds );

                        //Also put back the cells PeriodEra[] stuff
                        memcpy(ptr_cellera, PreMovePeriodEra, sizeof(PreMovePeriodEra) );

                        CopyToPre = FALSE;              //Will tell top of loop not to copy these values back
into the Pre...[], they are already there!
                }
        }
// ====================== End of Great Deluge acceptance routine ==============================

}//end for(loop...)
printf("\n");       //because of the way I have the print statements for my viewing

//*****************************************************************************************

//Copy the BestSolution back over to Solution & The BestEra back to S_Era so everything is kosher
memcpy( CS, Best, sizeof(struct SOLUTION) * count);
memcpy( S_Era, BestEra, sizeof(struct ERA) * SolutionSheds);

//close files
fclose(WRITE_BEST);
fclose(WRITE_RANDOM);

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\t\t****** GreatDeluge for goal %d took %.2lf seconds******\n", GOAL_TO_USE,Duration );



//Testprint
printf("The best solution found has ERA values of:\n");
for(a=0;a<(signed)SolutionSheds;a++)
{
        printf("Subwatershed %hu has Count %lu:   ",S_Era[a].Minor, S_Era[a].Count);

        for(b=0;b<NP;b++)
                printf("\t%.2f",((float)S_Era[a].SumPeriodEra[b] / ERA_EXP) / S_Era[a].Count);

        printf("\n");

}

//*****************************************************************************
//                                  Double check

printf("\n\n========== Getting ready to DOUBLE CHECK solution ==============================\n");
//Resort the array of SOLUTION structures by MINOR .  This will guarantee all the subwatersheds are in order
//Use mgsort because qsort takes way too long since there are not many unique Minor ID's
mgsort(   (void*)CS,                                    //base
                (size_t)count,                                          //count of # of arrays
                sizeof(struct SOLUTION),                        //size of each array
                0, count-1,                                                     //current division (
always: 0, "Count"-1 )
                LookAtSolutionMinor );                                  //compare function

//Initialize this array of ERA structures - this is important because Fill_SEra will do some += summing
memset( S_Era, 0, sizeof(struct ERA) * SolutionSheds );

//Fill the array of S_Era structures with appropriate values
if( Fill_SEra(SolutionSheds, S_Era, count, CS) == FALSE )
        Bailout(85);

//Resort the  array of CS structures by CELLID only. This is needed by CalculateSumPeriodEra
qsort(    (void*)CS,                                                    //base
                (size_t)count,                                                                  //count of # of
arrays
                sizeof(struct SOLUTION),                                                //size of each array
                LookAtSolutionCellid          );                                        //compare
function

if( CalculateSumPeriodEra(SolutionSheds, S_Era, count, CS, OV, Records) == FALSE )
        Bailout(86);

printf("==========================  Finished double checking ==============================\n\n");
//                                  End DoubleCheck
//*************************************************************************

//********************** These will print out ONLY those things for cells IN THE SOLUTION *****************
```

```
//Print out the Periodic values for the Value being optimized
PrintSolutionValues(count, CS, OV, Records, SubEra, ACTUAL);

//Print out the Big Trees
PrintSolutionBigTrees(count, CS, OV, Records, SubEra, ACTUAL);

//Print out the ERA values in S_Era
PrintSolutionEraValues(S_Era,SolutionSheds, ACTUAL);


//Delete stuff on the free store
delete [] PreMoveEra;
delete [] BestEra;
delete [] Best;

        return TRUE;
}//end DelugeGoal1
```

GOALS.H

```
//Mostly some functions defined in GOAL_CONTROLLER.CPP and are used by almost all the GOAL*.cpp files.


extern int FillValueToOptimize(void);


extern double GetBaselineVTO(ulong count, struct SOLUTION Solution[], double PerValues[],
                                                                                    struct
OPTIMIZE_SINGLE_VALUE OV[], ulong Records);

extern void SwapAdjust( struct SOLUTION *ptr_cs, ushort NG, ushort NH, double PerValues[], struct
OPTIMIZE_SINGLE_VALUE OV[],
                                    ulong Records, struct ERA *ptr_era);

extern void PrintSolutionValues(ulong Count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[],
                                                    ulong Records, int SubEra[], int Status);

extern void PrintSolutionBigTrees(ulong Count, struct SOLUTION CS[], struct OPTIMIZE_SINGLE_VALUE OV[],
                                                    ulong Records, int SubEra[], int Status);

extern int DetermineEligibleCells(ulong Values[]);

extern int FillSolution(ulong Values[], struct SOLUTION Solution[], int Status);

extern int LookAtOSV(const void *ptr1, const void *ptr2);

extern int CompareEraMinor(const void *ptr1, const void *ptr2);

extern int LookAtSolutionMinor(const void *ptr1, const void *ptr2);

extern int LookAtSolutionCellid(const void *ptr1, const void *ptr2);

extern ulong CountSolutionWatersheds(ulong count, struct SOLUTION Solution[]);

extern int Fill_PValues(ulong Count, struct SOLUTION CS[], ulong Records, struct OPTIMIZE_SINGLE_VALUE OV[], double
Value[] );

extern int InputSolution(ulong Count, struct SOLUTION CS[] );

extern void InputAndCalculateSolutionEras(ulong Count, struct SOLUTION CS[] );

extern void PrintSolutionEraValues(struct ERA Era[], ulong NoSheds, int Status);

extern int Fill_SEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[] );

extern int FillEndingEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[] );

extern void OutputInitialGoal(void);

//this is really defined in EraStuff.cpp
extern int CalculateSumPeriodEra(ulong NoSub, struct ERA S_Era[], ulong Count, struct SOLUTION CS[], struct
OPTIMIZE_SINGLE_VALUE OV[],

                                                    ulong Records);
```

MGSORT.CPP

```
/*****************************************************************************
*                                                                            *
*  ------------------------------ mgsort.c ------------------------------    *
*                                                                            *
*****************************************************************************/

#include <stdlib.h>
#include <string.h>


int mgsort(void *data, int size, int esize, int i, int k, int (*compare)
   (const void *key1, const void *key2));
```

```
static int merge(void *data, int esize, int i, int j, int k, int (*compare)
   (const void *key1, const void *key2));

//-------------------------------- End of function definitions ------------------


/*****************************************************************************
*                                                                            *
*  ------------------------------ mgsort ------------------------------      *
*                                                                            *
*****************************************************************************/
int mgsort(void *data, int size, int esize, int i, int k, int (*compare)
   (const void *key1, const void *key2))
//***************************************************************************
{

int               j;

/*****************************************************************************
*                                                                            *
*  Stop the recursion when no more divisions can be made.                    *
*                                                                            *
*****************************************************************************/

if (i < k) {

   /*****************************************************************************
   *                                                                            *
   *  Determine where to divide the elements.                                   *
   *                                                                            *
   *****************************************************************************/

   j = (int)(((i + k - 1)) / 2);

   /*****************************************************************************
   *                                                                            *
   *  Recursively sort the two divisions.                                       *
   *                                                                            *
   *****************************************************************************/

   if (mgsort(data, size, esize, i, j, compare) < 0)
      return -1;

   if (mgsort(data, size, esize, j + 1, k, compare) < 0)
      return -1;

   /*****************************************************************************
   *                                                                            *
   *  Merge the two sorted divisions into a single sorted set.                  *
   *                                                                            *
   *****************************************************************************/

   if (merge(data, esize, i, j, k, compare) < 0)
      return -1;

}

return 0;

}



/*****************************************************************************
*                                                                            *
*  ------------------------------ merge -------------------------------      *
*                                                                            *
*****************************************************************************/
static int merge(void *data, int esize, int i, int j, int k, int (*compare)
   (const void *key1, const void *key2))
//***************************************************************************
{

char              *a = (char *)data,
                  *m;

int               ipos,
                  jpos,
                  mpos;

/*****************************************************************************
*                                                                            *
*  Initialize the counters used in merging.                                  *
*                                                                            *
*****************************************************************************/

ipos = i;
jpos = j + 1;
mpos = 0;

/*****************************************************************************
*                                                                            *
```

```
*  Allocate storage for the merged elements.                                *
*                                                                           *
*****************************************************************************/

if ((m = (char *)malloc(esize * ((k - i) + 1))) == NULL)
   return -1;

/*****************************************************************************
*                                                                            *
*  Continue while either division has elements to merge.                     *
*                                                                            *
*****************************************************************************/

while (ipos <= j || jpos <= k) {

   if (ipos > j) {

      /*************************************************************************
      *                                                                        *
      *  The left division has no more elements to merge.                      *
      *                                                                        *
      *************************************************************************/

      while (jpos <= k) {

         memcpy(&m[mpos * esize], &a[jpos * esize], esize);
         jpos++;
         mpos++;

      }

      continue;

      }

   else if (jpos > k) {

      /*************************************************************************
      *                                                                        *
      *  The right division has no more elements to merge.                     *
      *                                                                        *
      *************************************************************************/

      while (ipos <= j) {

         memcpy(&m[mpos * esize], &a[ipos * esize], esize);
         ipos++;
         mpos++;

      }

      continue;

   }

   /**************************************************************************
   *                                                                         *
   *  Append the next ordered element to the merged elements.                *
   *                                                                         *
   **************************************************************************/

   if (compare(&a[ipos * esize], &a[jpos * esize]) < 0) {

      memcpy(&m[mpos * esize], &a[ipos * esize], esize);
      ipos++;
      mpos++;

      }

   else {

      memcpy(&m[mpos * esize], &a[jpos * esize], esize);
      jpos++;
      mpos++;

   }

}

/*****************************************************************************
*                                                                            *
*  Prepare to pass back the merged data.                                     *
*                                                                            *
*****************************************************************************/

memcpy(&a[i * esize], m, esize * ((k - i) + 1));

/*****************************************************************************
*                                                                            *
*  Free the storage allocated for merging.                                   *
*                                                                            *
*****************************************************************************/

free(m);
```

```
return 0;

}
```

PREPAREFARSITE.CPP

```
//*************************************************************************************************************
//This PrepareFarsite.cpp file contains the functions that are used to prepare and run farsite within this
//SAFED program.
//*************************************************************************************************************
//*************************************************************************************************************


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"
#include "data.h"


//Functions defined here in PrepareFarsite.cpp
int PrepareFarsite(int period, int weather);

FILE *open_input(char *filename, int rw);
int close_file(FILE *f,  char *filename);

void InitialFiles(void);
void RunMakeper0(void);

void MakeRunfarsite(int p);
void WhichOutputs(int p);
void MakeIgnition(int p);
void MakeFiresl(int p);
void prepare_run(int p, int TotalHours);
int IgnitionPoints(int p, int weather);
void PrepareFarsiteEnvt(int p, int drought);
void PrefireInfo(int p, int Hours, int Drought, int NoFires);



//*************************************************************************************************
int PrepareFarsite(int period, int weather)
//*************************************************************************************************
{
                    // ***IMPORTANT **  Remember that the LayerFile is made in PrepareFlammap() now, they both use
same file

//Some local variables
int Hours, NoFires=0;


// Use to run all the PrepareFarsite functions


//Set the hours to burn as follows:
if(weather == 1)
          Hours = 24;                              //Wet
else if(weather == 2)
          Hours = 48;                              //Moderate
else
          Hours = 96;                              //Drought..weather is either 3 or 4 (mild or severe drought)

                    MakeRunfarsite(period);                                                    //Make the
runit.bat file needed for each period

                    WhichOutputs(period);                                                 //Make the "Farout" file
that specifies which output

          //raster grids to make.

                    MakeIgnition(period);                                                 //Make the ignition.txt
file needed for each period

                    MakeFiresl(period);                                                   //Make the firesl.txt
file needed for each period

#ifndef RERUN_SIM
                    prepare_run(period,Hours);                                  //prepare run.txt to change the
start dates and how

          //long the fire burns

                    NoFires = IgnitionPoints(period, weather);                  //Create the ignition points needed
for this period
#endif

                    PrepareFarsiteEnvt(period,weather);                         //Create the envt.txt file so that
we can make the

          //necessary changes to it before it is used
```

```
                PrefireInfo(period, Hours,weather, NoFires);



return TRUE;
}//end of PrepareFarsite


//*****************************************************************************************************************
FILE *open_input(char *filename, int rw)
//*****************************************************************************************************************
{
        FILE *f;
        errno = 0;
        char mode[2];


        if(rw == 1) {mode[0] = 'r'; mode[1] = '\0';}
        if(rw == 2) {mode[0] = 'w'; mode[1] = '\0';}

        if(filename == NULL) filename = "\0";
        f = fopen(filename,mode);
        if (f == NULL)
                fprintf(stderr, "open_input(\"%s\") failed:  %s\n", filename, strerror(errno));

        return f;
}

//*****************************************************************************************************************
int close_file(FILE *f,  char *filename)
//*****************************************************************************************************************
{
        int s = 0;
        if (f == NULL) return 0;
        errno = 0;
        s = fclose(f);
        if (s == EOF) perror("Close failed");

        return s;
}


//*****************************************************************************************************************
void InitialFiles(void)
//*****************************************************************************************************************
{

        //This is to create the Batch file needed to change the directory and start process to pipe
        //a file through Arc, which in turn will run the make_per0.aml.

        //Will create two files.  1) a file called make_per0.bat and the other called make_per0.txt
        //which have a specific format as seen below.

        //Don't need the period passed to this function because this will only be executed once
        //and it will automatically put it in the per0 directory

        FILE *OpenWrite;
        char WriteOut[150];
        char StartArc[150];
        char ArcCommand[350];

//------------------------------------- End variable defining ---------------------------------------

        //stuff for .bat file
        char DirChange[100];
        sprintf(StartArc, "type %s%s%d\\per0\\make_per0.txt | arc",PREFIX,INPUTS,GOAL_TO_USE);

        //stuff for the .txt file
        sprintf(ArcCommand,  "&r %s%s\\make_per0.aml %s %s %d",PREFIX,AmlDir, MAIN_USER, ENVT, FILE_TYPE);

        //Prepare and write the make_per0.bat file and a command line to change directories
        sprintf(WriteOut, "%s%s%d\\per0\\make_per0.bat",PREFIX,INPUTS,GOAL_TO_USE);
        sprintf(DirChange, "cd %s%s%d\\per0\\",PREFIX,INPUTS,GOAL_TO_USE);


                OpenWrite = open_input(WriteOut, 2);                                    //open in write
mode
                fprintf(OpenWrite, "%s\n", DirChange);
                fprintf(OpenWrite, "%s\n", StartArc);
                close_file(OpenWrite, WriteOut);

        //Prepare and write the make_per0.txt file
        sprintf(WriteOut, "%s%s%d0\\make_per0.txt",PREFIX,INPUTS,GOAL_TO_USE);

                OpenWrite = open_input(WriteOut, 2);                                    //open in write
mode
                fprintf(OpenWrite, "%s\n", ArcCommand);
                close_file(OpenWrite, WriteOut);



        //Call up the RunMakeper0() to actually start the batch file
```

```
        RunMakeper0();
}

//*****************************************************************************************************************
void RunMakeper0(void)
//*****************************************************************************************************************
{
        // This function will run the MAKE_PER0.BAT file which in turns calls up ArcInfo and then
        // runs the make_per0.aml located in the g:/Model/amls/  directory.  In essence, the initial
        //grid files needed before period 1 starts will be created and put into either the ./per0/
        //or ..\inputs\Constant\* directory, so that when per1 starts it can look in that directory.

        //For Time information
clock_t Start, Finish;
double Duration;


Start = clock();

        char RunBatch[100];

        //Make the command
        sprintf(RunBatch, "%s%s%d0\\make_per0.bat",PREFIX,INPUTS,GOAL_TO_USE);

        system(RunBatch);

Finish = clock();
Duration = ( (double)(Finish-Start) / CLOCKS_PER_SEC );
printf("\n**It took %.2lf seconds to run the MakePer0.aml**\n", Duration );


}


//*****************************************************************************************************************
void MakeRunfarsite(int p)
//*****************************************************************************************************************
{
        //Make the RUNFARSITE.BAT file needed by Farsite - on the fly

        FILE *OpenWrite;
        char WriteOut[150];

        char LayerFile[150];
        char EnvtFile[150];
        char IgnitionFile[150];
        char RunFile[150];
        char BarrierFile[150];
        char ChangesFile[150];
        char OutputFiles[150];

//-------------------------------------- End variable defining ---------------------------------------

        //String together the current period directory path and the appropriate file names
        sprintf(WriteOut,           "%s%s%d\\per%d\\runfarsite.bat",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(LayerFile,          "%s%s%d\\per%d\\layers.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(EnvtFile,           "%s%s%d\\per%d\\farsite_envt.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(IgnitionFile,       "%s%s%d\\per%d\\ignition.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(RunFile,            "%s%s%d\\per%d\\run.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(OutputFiles,        "%s%s%d\\per%d\\Farsite_out.txt",PREFIX,OUTPUTS,GOAL_TO_USE,p);
        sprintf(BarrierFile,        "%s%s\\barrier.txt",PREFIX,ConstantInput);
        sprintf(ChangesFile,        "%s%s\\changes.txt",PREFIX,ConstantInput);



        OpenWrite = open_input(WriteOut, 2);                                    //open in write mode
        fprintf(OpenWrite,"%s%s %s %s %s %s %s %s %s %s\n",PREFIX,FarsiteName,LayerFile,EnvtFile,

                IgnitionFile,RunFile,OutputFiles,BarrierFile,ChangesFile);

                close_file(OpenWrite, WriteOut);

}
//*****************************************************************************************************************
void WhichOutputs(int p)
//*****************************************************************************************************************
{
        // This function will make a file called "...outputs\per*\Farsite_out.txt".  Inside of this file
        //will be a list of the raster grids we want Farsite to output.  This file is specified in the
        //above runit.bat file as the 6th parameter that Farsite is looking for and which the Farsite code
        //calls the "OutputFile".  In farsite, I have modified the code in Farread.cpp
ReadInputFiles::ReadRunSpecs
        //to strip off the last 16 chararcters ( \Farsite_out.txt ) and then search for the first occurance
        //of "per" and set a pointer at the beginning of "per" to pass along and create output filenames
        //such as "per1", "per2", ..., "per20".

        //All we want for now are the FLAME lengths and unless we need more that is all that is going in.
        //The arrival grid will ALWAYS be made because that is needed for Farsite to do its calculations.


        FILE *OpenWrite;
        char WriteOut[50];
        char Grid1[10] = "flame";
        char Temp[150];
//-------------------------------------- End variable defining ---------------------------------------
```

```
        //Make the correct filename
        sprintf(Temp, "%s%s%d\\per%d\\Farsite_out.txt",PREFIX,OUTPUTS,GOAL_TO_USE,p);
        sprintf(WriteOut, "%s",Temp);

        OpenWrite = open_input(WriteOut, 2);                                    //open in write mode
                fprintf(OpenWrite,"%s\n",Grid1);


        close_file(OpenWrite, WriteOut);

}



//*****************************************************************************************************************
void MakeIgnition(int p)
//*****************************************************************************************************************
{

        //Make the ignition.txt file which specifies to use fires1.txt as the file that has the name of
        //the ascii files which have the coordinates for fire ignition sources  - this is all screwy but
        //that is how Farsite is coded and I have just followed along


        FILE *OpenWrite;
        char WriteOut[100];
        char TextLine[100];


        //Cat together the full directory path name and the file names
        sprintf(TextLine, "%s%s%d\\per%d\\fires1.txt",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(WriteOut, "%s%s%d\\per%d\\ignition.txt",PREFIX,INPUTS,GOAL_TO_USE,p);


#ifdef DEBUG_MAKEIGNITION
        //print out to see if they are correct
        printf("%s\n",TextLine);
#endif


                OpenWrite = open_input(WriteOut, 2);                                    //open in write
mode
                        fprintf(OpenWrite,"%s\n",TextLine);
                close_file(OpenWrite, WriteOut);

}

//*****************************************************************************************************************
void MakeFires1(int p)
//*****************************************************************************************************************
{

        //Make the fires1.txt file which specifies the potential three ascii files that can be populated
        //With data concerning fire ignition sources.  They must be in the specified order because that
        //is how Farsite is looking for them


        FILE *OpenWrite;
        char WriteOut[50];
        char Points[100], Lines[100], Poly[100];

//------------------------------------- End variable defining ---------------------------------------

        //Make the full directory path name and the file names
        sprintf(Points, "%s%s%d\\per%d\\igpoints.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(Lines, "%s%s%d\\per%d\\iglines.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(Poly, "%s%s%d\\per%d\\igpoly.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
        sprintf(WriteOut, "%s%s%d\\per%d\\fires1.txt",PREFIX,INPUTS,GOAL_TO_USE,p);


                OpenWrite = open_input(WriteOut, 2);                                    //open in write
mode
                        fprintf(OpenWrite,"%s\n",Points);
                        fprintf(OpenWrite,"%s\n",Lines);
                        fprintf(OpenWrite,"%s\n",Poly);
                close_file(OpenWrite, WriteOut);

}

//*****************************************************************************************************************
void prepare_run(int p, int TotalHours)
//*****************************************************************************************************************
{

        FILE  *OpenWrite;
        char RunFile[100];


        char TimeStep[5]="0400";                        // Change this value if needed to speed up processing
                                                                                //May want to
come up with some sort of ratio of Time
                                                                                //step to the
TotalHours because this really slows us
```

```
                                                                                                    //down with
long fires.


        int PerimeterRes = 200;                                         // Change these two to speed up if coarser
res. is OK
        int DistanceRes = 200;
        char RasterUnits[10]="metric";
        int  MonthStart, DayStart, HourStart;
        int    MonthEnd, DayEnd,   HourEnd;
        int rnd;
        int DaysOfBurning, HoursLeft, MaxDay ;
        int Xday, VisStep;

//------------------------------------ End variable defining ---------------------------------------------

        //Create a string with the actual run.txt file name with the full directory path
        sprintf(RunFile, "%s%s%d\\per%d\\run.txt",PREFIX,INPUTS,GOAL_TO_USE,p);


//*************************************************************************************************
                /*  We are going to generate new
                dates and times according to the following process:  The main program will send a
                value to this function that is the # of hours to burn. The burn start month will be
                randomly decided as either June or July.  A random day and hour will then be picked and the
                hours to burn will be added and the end hour, day, and month will be calculated.*/
//*************************************************************************************************


                //Randomly generate number to determine if start month is june or july
                rnd = (rand() % 2);

                //NEW- Bernie changed wind&weather files to have only one month of data Aug15-Sept 14
                //so make it always start in August (31 day month)      (30 April 99)
                MonthStart = 8;

                //if (rnd == 0) MonthStart = 6;                                 //for June
                //else MonthStart = 7;                                                  //for July



                //Randomly generate number to get which day of the month it is
                //NEW(30April99) - must start on or after the 15th because of new .wnd and .wtr files

                /*if (MonthStart == 6)  rnd = (rand() % 30 + 1);    //will get 0-29 and add 1 to get 1-30 days
in June
                else                                            rnd = (rand() % 31 + 1);     //will get 0-30 and add 1
to get 1-31 days in July or August
                */
                if (MonthStart == 6)
                {
                        do{
                                rnd = (rand() % 30 + 1);
                        }while(rnd < 16);                                       //Farsite sometimes bail if start
date is the 15th - maybe because that is first line?
                }
                else    //only for July and August since they both have 31 days. Modify if other months
added!
                {
                        do{
                                rnd = (rand() % 31 + 1);
                        }while(rnd < 16);
                }
                DayStart = rnd;

                //Randomly generate number to get the start hour
                //NEW - Bernie wants fire to always start from 0800 to 2000 hours
                //and they should start on Even hours to match weather and wind files(30 April 99)
                do
                {
                rnd = (rand() % 20 + 1);
                }while(rnd < 8 || (rnd % 2) != 0);

                HourStart = rnd;

                //Determine how many days and left over hours there are from TotalHours
                DaysOfBurning = (int)(TotalHours / 24);
                HoursLeft = (int)(TotalHours % 24);

                //Set MaxDay and MaxHours to make sure days and hours added to start date don't exceed a valid
month
                switch(MonthStart)
                {
                case 6: MaxDay=30;  break;              //days in June
                case 7: MaxDay=31;  break;              //days in July
                case 8: MaxDay=31;  break;              //days in August
                }

                //Calculate the ending Hours, days, and Month
                Xday = 0;
                HourEnd = HourStart + HoursLeft;
                if(HourEnd > 24)
                {
                        Xday = HourEnd / 24;                    // if more than 24 hours make another day
```

```
                        HourEnd = HourEnd % 24;                 //otherwise this is the final hour
                }

                DayEnd = DayStart + DaysOfBurning + Xday;       //set ending day
                if(DayEnd > MaxDay)                             //if days burning exceed allowable #days in a
month
                {
                        MonthEnd = MonthStart + 1;      //increment to next month
                        DayEnd = DayEnd % MaxDay;       //reset ending day
                }
                else
                        MonthEnd = MonthStart;                  //otherwise fire ends in same month

                //Set the Visible Time Step equal to that of the TOTALHOURS time
                VisStep = TotalHours;


//*************************************************************************************************
                //Prepare all the data to write back into a run.txt file so that it can be used by FARSITE
                //It has to be written  in this exact format otherwise Farsite will bomb out.
//*************************************************************************************************

                OpenWrite = open_input(RunFile, 2);

                fprintf(OpenWrite, "1998%.2d%.2d%.4d\n",MonthStart,DayStart,(HourStart*100));
                fprintf(OpenWrite, "1998%.2d%.2d%.4d\n",MonthEnd,DayEnd,(HourEnd*100));
                fprintf(OpenWrite, "%s\n",TimeStep);
                fprintf(OpenWrite, "%.4d\n",(VisStep*100));
                fprintf(OpenWrite, "%d\n",PerimeterRes);
                fprintf(OpenWrite, "%d\n",DistanceRes);
                fprintf(OpenWrite, "%s\n",RasterUnits);

                close_file(OpenWrite, RunFile);

}               //end of prepare_run

//*************************************************************************************************************
int IgnitionPoints(int p, int weather)
//*************************************************************************************************************
{

/*
This function will generate fire ignition points for each period.  The output will be a file called
...\inputs\\per*\igpoints.asc  which FARSITE will use.  The format of the file is standard ARC/Info "ungenerate"
form:

  PointID X-coord(in meters)            y-coord(meters)
  ...
  ...
  End

NOTE:  The data stored in Data.*[][p] is that BEFORE any fire so that data is GOOD.  After the fire
that information will be updated with new condition and can then be sent out for mapping, etc..
*/


int NoFires, NewPoint = 0;
int a,b, Row,Column, ContinueStatus, AnotherContinueStatus, EvalColumn;
FILE  *OpenWrite;
char WriteOut[150];
ushort Points[15][2], RowsAway, ColumnsAway;
ulong *ptr_treelist;
int rnd;
float XValue, YValue;


int *ptr_srp;                   //Starting Row Position
ushort   *ptr_gridcolumn, *ptr_veg, *ptr_fuel, *ptr_elev, *ptr_fire;
int r,c,HowMany;

//-------------------------------------- End variable defining ---------------------------------------------

//Initialize Points[][] which will hold the ROW and COLUMN value for selected ignition points
for(a=0;a<15;a++)
{
        for(b=0;b<2;b++)
                Points[a][b] = 0;
}


//CHANGE: 7 June 99:  Bernie now wants to have 5 - 15 ignition points regardless of
//whether it is a Wet, Moderate, or Drought period.  The weather and wind files will adjust for conditions
//14 March 00 - changing to max of 14 - FARSITE sometimes "hangs" with 15 fires????
do
{
        NoFires = (rand() % 14 + 1);
}while(NoFires < 5 );

        do
        {
                Row                 = ( rand() % ROWS + 1 );        //get 1 - ROWS
                Column     = ( rand() % COLUMNS + 1 ); //get 1 - COLUMNS

                rnd = ( rand() % 100 + 1);                      //assign random number to use for
probabilistic comparisons later
```

```
                    //printf("Row is %d and Column is %d\n",Row,Column);

                    //************************************  CHECKER #1
**********************************************
                    //======================== Check for WATER condition and any other immediate disqualifiers
============
                    ptr_srp =                       &link[Row-1][1];
                    HowMany =                       *(ptr_srp+1);
                    ptr_gridcolumn =    &Data.GridColumn[(*ptr_srp)-1];
                    ptr_treelist =                  &Data.Treelist[(*ptr_srp)-1];
                    ptr_veg =                       &Data.InitialVeg[(*ptr_srp)-1];
                    ptr_elev =                              &Data.Elev[(*ptr_srp)-1];
                    ptr_fire =                              &Data.FireHistory[(*ptr_srp)-1];
                    ptr_fuel =                              &Data.FuelModel[(*ptr_srp)-1][p-1];

                    //If the whole row is blank, pick another point
                    if( *ptr_srp == FALSE )                 //means a zero was left in this spot during MakeLink
                              continue;

                    ContinueStatus = 0;
                    for(c=0;c<HowMany;c++)
                    {
                              if( *ptr_gridcolumn == Column)                    //This is the correct cell
                              {
                                        // ******* Do any CELL SPECIFIC checking below, using the same format
**********

                                        //Check to see if the cell location was classified as water in the Initial
Vegeation classification
                                        if( *ptr_veg == GIS_WATER)
          //Yes it was
                                        {
                                                  ContinueStatus = 1;
          //FAIL, try another point
                                                  if( *ptr_fuel != 98 || *ptr_treelist != NONFOREST)
          //should match with Fuel model 98 that was inputted in ReadData.cpp
                                                            Bailout(54);
                                                  break;                            //finished looking in this for...
loop
                                        }


                                        //Check to see if this cell is in or out of the polygons that delineate previous fire
history.  If it
                                        //is out, give it a 40% chance of continuing because there are no fire history
polygons in the
                                        //NW portion of the Applegate and that would be crazy to exclude fire from starting
there.
                                        //And only do this when not running with TINY or COMPARE watersheds (they are too
small and no points get picked)
#if !defined(TINY_RUN)  && !defined(COMPARE_RUN)
                                        if( *ptr_fire == NODATAFLAG)              //is outside any fire history
polygons
                                        {
                                                  if( rnd > 60)
          //give it a 60% chance of occuring anyways
                                                  {
                                                            ContinueStatus = 1;                     //Fail, try
another point
                                                            break;
          //finished looking in this for...loop
                                                  }
                                        }
#endif

                                        //Get a new random number
                                        rnd = ( rand() % 100 + 1);

                                        //put a probability factor in here to account for weather and elevation
liklihood
                                        //(i.e. in Wet years, there is a smaller probablity that fires will occur
in high elevation)
                                        //get those values from Bernie - this method may not be the most accurate
because new
                                        //random numbers are compared against new random points and it could cycle
enough that
                                        //ignition points get located in undesirable elevations just because of
cycling.
                                        if(weather == 1)
          //wet period
                                        {
                                                  if( *ptr_elev >= (3000*FT2M) )
                                                  {
                                                            if( rnd > 10)
                              //only 10% chance that this is allowable
                                                            {
                                                                      ContinueStatus = 1;
          //Fail
                                                                      break;
                                        //finished looking in this for... loop
                                                            }
                                                  }
                                                  else if( *ptr_elev >= (1500*FT2M) )
                                                  {
```

```
                                                  if( rnd <= 10 || rnd > 25)
//only 15% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                                        else
                        //less than 1500 feet in elevation
                                        {
                                                  if( rnd <= 25)
          //a 75% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                              }
                              else if(weather == 2)
          //moderate period
                              {
                                        if( *ptr_elev >= (3000*FT2M) )
                                        {
                                                  if( rnd > 15)
          //only 15% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                                        else if( *ptr_elev >= (1500*FT2M) )
                                        {
                                                  if( rnd <= 15 || rnd > 50)
//only 35% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                                        else
                        //less than 1500 feet in elevation
                                        {
                                                  if( rnd <= 50)
          //a 50% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                              }
                              else
                        //drought period - either Mild or Severe
                              {
                                        if( *ptr_elev >= (3000*FT2M) )
                                        {
                                                  if( rnd > 10)
          //only 10% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                                        else if( *ptr_elev >= (1500*FT2M) )
                                        {
                                                  if( rnd <= 10 || rnd > 55)
//only 45% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
                                                  }
                                        }
                                        else
                        //less than 1500 feet in elevation
                                        {
                                                  if( rnd <= 55)
          //a 45% chance that this is allowable
                                                  {
                                                            ContinueStatus = 1;
//Fail
                                                            break;
                        //finished looking in this for... loop
```

```
                                                                }
                                                        }
                                                }//end of checking the weather and elevation

                                        }//end if( *ptr_gridcolumn == Column)

                                        ptr_gridcolumn++;
                                        ptr_treelist++;
                                        ptr_veg++;
                                        ptr_elev++;
                                        ptr_fire++;
                                        ptr_fuel+=NP;

                                }//end for(c=0;c<HowMany;c++)

                                if( ContinueStatus == 1)
                                        continue;                                                       //get another
point

                                //************************************  CHECKER #2
*********************************************
                                //Now check and make sure this cell is not within a 1 mile "buffer" of the edge.  Do this by
ensuring
                                //there is a miles worth of contiguous cells (MOC) BEFORE, AFTER, ABOVE, and BELOW the selected
cell.

                                //NOTE: All I'm checking here is to see if there is a valid cell in Grid.Gridcolumn - that DOES
NOT
                                //ensure there is data for all themes (i.e. fuel, etc.), only that there was NOT nodata there
when
                                //the Cellid.asc file was brought in.  Check later to see if there is data for necessary
themes.

                                if(Row <= MOC)                                          //the starting row is too close to begin
with, pick another point
                                        continue;
                                if(Row >= (COLUMNS - MOC) )
                                        continue;                                       //starting row is too close to bottom of
area, pick another point
                                if(Column <= MOC)
                                        continue;                                       //starting column is too close to left edge
of data
                                if(Column >= (ROWS - MOC) )
                                        continue;                                       //starting column is too close to right edge
of data

                                //we know the current evaluation row, look in link[][] and see if there is even any data in
that row
                                if(link[Row-1][2] == 0)                 //there is no data for this row - occurs above or below
the geo. extent of current envt.!
                                        continue;                                               //so pick another point

                                        //******************   Look ABOVE the cell for MOC valid cells ******************
                                //printf(" ********** Looking ABOVE the cell for firepoint #%d
******************\n",NewPoint+1);
                                ContinueStatus = 0;
                                for(r=Row-1;r>=Row-MOC;r--)
                                {
                                        //printf("Selected ROW is %d and COLUMN is %d....and now evaluating row
%d",Row,Column,r);

                                        //we know the current evaluation row, get its SRP in link[][] and check that rows
Data.Gridcolumn values
                                        ptr_srp = &link[r-1][1];
                                        HowMany = *(ptr_srp+1);

                                        if(HowMany == 0)                //there are no columns of data for this row - bad!
                                        {
                                                //printf("...which is FALSE because of lack of LINES\n");
                                                ContinueStatus = 1;
                                                break;                                                  //quit looking above
because there are a shortage of lines above
                                        }

                                        //Set pointer where this grid row starts in the Data.* arrays
                                        ptr_gridcolumn = &Data.GridColumn[(*ptr_srp)-1];
                                        AnotherContinueStatus = 0;
                                        for(c=0;c<HowMany;c++)
                                        {
                                                if( *ptr_gridcolumn == Column)
                        //YES, this row does have a cell in the same column
                                                {
                                                        //printf("...which is TRUE\n");
                                                        AnotherContinueStatus = 1;
                                                        break;
                        //quit looking at this row in Data.Gridcolumn
                                                }
                                                ptr_gridcolumn++;
                                        }

                                        if(AnotherContinueStatus == 1)
                                                continue;                                               //next
iteration of For(r=Row-1...} - check next cell above
                                        else
                                        {
```

```
                                        //printf("...which is FALSE because there was NODATA above\n");
                                        ContinueStatus = 1;
                                        break;                                                                  //no
column above was found
                                }

                        }//end for(r=Row-1;r>=Row-64;r--)

                        if(ContinueStatus == 1)                                         //failed to have MOC above that
column, try another point
                                continue;
                        // ********************** End of looking ABOVE the current cell
*********************************************

                        //NOTE: only gets here if everything above passes

                        //******************   Look BELOW the cell for MOC valid cells ******************
                        //printf(" ********** Looking BELOW the cell for firepoint #%d
******************\n",NewPoint+1);
                        ContinueStatus = 0;
                        for(r=Row+1;r<=Row+MOC;r++)
                        {
                                //printf("Selected ROW is %d and COLUMN is %d....and now evaluating row
%d",Row,Column,r);

                                //we know the current evaluation row, get its SRP in link[][] and check that rows
Data.Gridcolumn values
                                ptr_srp = &link[r-1][1];
                                HowMany = *(ptr_srp+1);

                                if(HowMany == 0)                //there are no columns of data for this row - bad!
                                {
                                        //printf("...which is FALSE because of lack of LINES\n");
                                        ContinueStatus = 1;
                                        break;                                                  //quit looking above
because there are a shortage of lines below
                                }

                                //Set pointer where this grid row starts in the Data.* arrays
                                ptr_gridcolumn = &Data.GridColumn[(*ptr_srp)-1];
                                AnotherContinueStatus = 0;
                                for(c=0;c<HowMany;c++)
                                {
                                        if( *ptr_gridcolumn == Column)
        //YES, this row does have a cell in the same column
                                        {
                                                //printf("...which is TRUE\n");
                                                AnotherContinueStatus = 1;
                                                break;
        //quit looking at this row in Data.Gridcolumn
                                        }
                                        ptr_gridcolumn++;
                                }

                                if(AnotherContinueStatus == 1)
                                        continue;                                                       //next
iteration of For(r=Row+1...) - check next cell below
                                else
                                {
                                        //printf("...which is FALSE because there was NODATA below\n");
                                        ContinueStatus = 1;
                                        break;                                                                  //no
column below was found
                                }

                        }//end for(r=Row+1;r<=Row+MOC;r++)

                        if(ContinueStatus == 1)                                         //failed to have MOC below that
column, try another point
                                continue;
                        // ********************** End of looking BELOW the current cell
*********************************************

                        //******************   Look BEFORE the cell for MOC valid cells ******************
                        //printf(" ********** Looking BEFORE the cell for firepoint #%d
******************\n",NewPoint+1);


                        //Set pointer where this grid row starts in the Data.* arrays
                        ptr_srp = &link[Row-1][1];
                        HowMany = *(ptr_srp+1);
                        ptr_gridcolumn = &Data.GridColumn[(*ptr_srp)-1];

                        //printf("Selected ROW is %d and COLUMN is %d....",Row,Column);

                        //Increment the pointer up to where the actual Data.Gridcolumn matches the current evaluation
column
                        ContinueStatus = 0;
                        for(r=0;r<HowMany;r++)
                        {
                                if( *ptr_gridcolumn == Column)
        //found the match - leave ptr_gridcolumn here
                                {
                                        ContinueStatus = 1;
                                        break;
```

```
                    }
                    ptr_gridcolumn++;
          }


          if(ContinueStatus != 1)                                   //This column, in this row, has
NODATA to begin with...pick another point
          {
                    //printf("...which is FALSE because this column has NODATA to begin with\n");
                    continue;
                                                                    //get another point
          }


          //otherwise, ptr_gridcolumn should be sitting on the right spot...check the continuity for MOC
BEFORE
          ContinueStatus = 0;
          for(c=1;c<=MOC;c++)
          {
                    //bump ptr_gridcolumn down the appropriate amount
                    ptr_gridcolumn--;

                    //get the evaluation column
                    EvalColumn = Column - c;

                    //printf("\n...now evaluating column %hu",EvalColumn);


                    if(*ptr_gridcolumn != EvalColumn)               //Discontinuity
                    {
                              //printf("...which is FALSE because there is NODATA here - DISCONTINUITY
BEFORE\n");
                              ContinueStatus = 1;
                              break;
                    }
          }

          if(ContinueStatus == 1)                                   //There was discontinuity
          {
                    //printf("...which is FALSE because column %d had nodata\n",EvalColumn);
                    continue;
                                                                    //get another point
          }

          //printf("...which is TRUE, there is continuity of data BEFORE\n");

          // ************************ End of looking BEFORE the current cell
*********************************************

          //NOTE: only gets here if everything above passes

          //******************   Look AFTER the cell for MOC valid cells ******************
          //printf(" ************ Looking AFTER the cell for firepoint #%d
******************\n",NewPoint+1);


          //Set pointer where this grid row starts in the Data.* arrays
          ptr_srp = &link[Row-1][1];
          HowMany = *(ptr_srp+1);
          ptr_gridcolumn = &Data.GridColumn[(*ptr_srp)-1];

          //printf("Selected ROW is %d and COLUMN is %d....",Row,Column);

          //Increment the pointer up to where the actual Data.Gridcolumn matches the current evaluation
column
          ContinueStatus = 0;
          for(r=0;r<HowMany;r++)
          {
                    if( *ptr_gridcolumn == Column)
        //found the match - leave ptr_gridcolumn here
                    {
                              ContinueStatus = 1;
                              break;
                    }
                    ptr_gridcolumn++;
          }


          if(ContinueStatus != 1)                                   //This column, in this row, has
NODATA to begin with...pick another point
          {
                    //printf("...which is FALSE because this column has NODATA to begin with\n");
                    continue;
                                                                    //get another point
          }

          //otherwise, ptr_gridcolumn should be sitting on the right spot...check the continuity for MOC
AFTER
          ContinueStatus = 0;
          for(c=1;c<=MOC;c++)
          {
                    //bump ptr_gridcolumn up the appropriate amount
                    ptr_gridcolumn++;
```

```
                        //get the evaluation column
                        EvalColumn = Column + c;

                        //printf("\n...now evaluating column %hu",EvalColumn);


                        if(*ptr_gridcolumn != EvalColumn)                //Discontinuity
                        {
                                //printf("...which is FALSE because there is NODATA here - DISCONTINUITY
AFTER\n");
                                ContinueStatus = 1;
                                break;
                        }
                }

                if(ContinueStatus == 1)                                  //There was discontinuity
                {
                        //printf("...which is FALSE because column %d had nodata\n",EvalColumn);
                        continue;
                                                                         //get another point
                }

                //printf("...which is TRUE, there is continuity of data AFTER\n");

        // ********************** End of looking AFTER the current cell
*****************************************

//================================= END OF ALL ERROR AND VALIDITY CHECKERS FOR THIS SELECTED CELL ==================


                //NOTE: only gets here if all CHECKERS have passed
                        Points[NewPoint][0] = Row;
                        Points[NewPoint][1] = Column;
                        NewPoint++;

        }while(NewPoint != NoFires);

//for(c=0;c<NoFires;c++)
        //printf("%hu\t%hu\n",Points[c][0],Points[c][1]);

//*************************************************************************************
//Now calculate actual coordinates for the points and send them out to a file for Farsite to read in.
//*************************************************************************************

//first, create the file to send the point data to
sprintf(WriteOut, "%s%s%d\\per%d\\igpoints.asc",PREFIX,INPUTS,GOAL_TO_USE,p);
OpenWrite = fopen(WriteOut, "w");

//loop through the Points[][] array and convert each point
for(c=0;c<NoFires;c++)                                  //There are only "NoFires" points stored in this array
(out of a possible 15)
{
        RowsAway =              ROWS - Points[c][0];                    //really the Y-offset from the
lower left corner of original grid
        ColumnsAway =       COLUMNS - Points[c][1];             //really the X-offset from the lower left
corner of original grid

        XValue = (float)F_XLL + (CELLSIZE * ColumnsAway);
        YValue = (float)F_YLL + (CELLSIZE * RowsAway);

        //print out the value
        fprintf(OpenWrite, "%d\t%.4f\t%.4f\n",c+1,XValue,YValue);
}//end for(c=0;c<NoFires;c++)

        //put in the final line in the igpoint.asc file
        fprintf(OpenWrite, "END\n");                                    //needed by Farsite because this is
ArcInfo "ungenerate" format


fclose(OpenWrite);

return NoFires;                 //so this can be passed on to PreFireInfo()

}//end IgnitionPoints()

//*************************************************************************************************************

//*************************************************************************************************************
void PrepareFarsiteEnvt(int p, int drought)
//*************************************************************************************************************
{
        //The farsite_envt.txt file called by Farsite specifies some files that Farsite will
        //use to set up the general parameters.  We are going to need to change things like
        //which weather and wind files it uses (based on whether it is a drought year or not).
        //There may be additional reasons to change certain files at a later time.

        //This file DOES NOT need to exist.  It will be created from scratch using the data from below.

        char custom[20]="CUSTOM_FUEL_FILE",                     CustomFuelFile[60];

        char conversion[20]="CONVERSION_FILE",          ConversionFile[60];

        char weather[20]="WEATHER_FILE",                        WetWeatherFile[60];
        char
        ModWeatherFile[60];
```

```
        char
        DroWeatherFile[60];

        char wind[20]="WIND_FILE",                                          WetWindFile[60];
        char
        ModWindFile[60];
        char
        DroWindFile[60];

        char fms[20]="FUELMOISTURE_FILE",                           WetFMFile[60];
        char
        ModFMFile[60];
        char
        DroFMFile[60];

        char adjustment[20]="ADJUSTMENT_FILE",          WetAdjustFile[60];
        char
        ModAdjustFile[60];
        char
        DroAdjustFile[60];

        char acceleration[20]="ACCELERATION_FILE",      AccelFile[60];

        char spot[20]="SPOT_FILE",                                          SpotFile[60];


        //Put together all the filenames
        sprintf(CustomFuelFile, "%s%s\\%s_farsite.fmd",PREFIX,ConstantInput,SHORT_NAME);

        sprintf(ConversionFile, "%s%s\\null.txt",PREFIX,ConstantInput);

        sprintf(WetWeatherFile, "%s%s\\%s_wet.wtr",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(ModWeatherFile, "%s%s\\%s_mod.wtr",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(DroWeatherFile, "%s%s\\%s_dro.wtr",PREFIX,ConstantInput,SHORT_NAME);

        sprintf(WetWindFile, "%s%s\\%s_wet.wnd",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(ModWindFile, "%s%s\\%s_mod.wnd",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(DroWindFile, "%s%s\\%s_dro.wnd",PREFIX,ConstantInput,SHORT_NAME);

        sprintf(WetFMFile, "%s%s\\%s_wet.fms",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(ModFMFile, "%s%s\\%s_mod.fms",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(DroFMFile, "%s%s\\%s_dro.fms",PREFIX,ConstantInput,SHORT_NAME);

        sprintf(WetAdjustFile, "%s%s\\%s_wet.adj",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(ModAdjustFile, "%s%s\\%s_mod.adj",PREFIX,ConstantInput,SHORT_NAME);
        sprintf(DroAdjustFile, "%s%s\\%s_dro.adj",PREFIX,ConstantInput,SHORT_NAME);

        sprintf(AccelFile, "%s%s\\null.txt",PREFIX,ConstantInput);
        sprintf(SpotFile, "%s%s\\spotting.txt",PREFIX,ConstantInput);

        //Use these to copy whichever of the above we want to a consistent output string name
        char WriteOut[250];
        char OutWeather[250];
        char OutWind[250];
        char OutMoisture[250];
        char OutAdjustment[250];

        FILE  *OpenWrite;


        //Now determine which of the files are going to be used
        if(drought == 1)                                                                    // is a WET
period
        {
                strcpy(OutWeather, WetWeatherFile);
                strcpy(OutWind, WetWindFile);
                strcpy(OutMoisture, WetFMFile);
                strcpy(OutAdjustment,WetAdjustFile);
        }
        else if(drought == 2)                                                               // is a
MODERATE period
        {
                strcpy(OutWeather, ModWeatherFile);
                strcpy(OutWind, ModWindFile);
                strcpy(OutMoisture, ModFMFile);
                strcpy(OutAdjustment,ModAdjustFile);
        }
        else
        {
                //is a DROUGHT period (Mild or Severe)
                strcpy(OutWeather, DroWeatherFile);
                strcpy(OutWind, DroWindFile);
                strcpy(OutMoisture, DroFMFile);
                strcpy(OutAdjustment,DroAdjustFile);
        }


        //Create a string with the actual envt.txt file name with the full directory path
        sprintf(WriteOut, "%s%s%d\\per%d\\farsite_envt.txt",PREFIX,INPUTS,GOAL_TO_USE,p);


                OpenWrite = open_input(WriteOut, 2);                                        //open in write
mode
                fprintf(OpenWrite, "%s \t%s\n", custom, CustomFuelFile );
                fprintf(OpenWrite, "%s \t%s\n", conversion, ConversionFile);
```

```
                fprintf(OpenWrite, "%s \t\t%s\n", weather, OutWeather);
                fprintf(OpenWrite, "%s \t\t%s\n", wind, OutWind);
                fprintf(OpenWrite, "%s \t%s\n", fms, OutMoisture);
                fprintf(OpenWrite, "%s \t%s\n", adjustment, OutAdjustment);
                fprintf(OpenWrite, "%s \t%s\n", acceleration, AccelFile);
                fprintf(OpenWrite, "%s \t\t%s\n", spot, SpotFile);
                close_file(OpenWrite, WriteOut);


}

//*****************************************************************************
void PrefireInfo(int p, int Hours, int Drought, int NoFires)
//*****************************************************************************
{
        // A little script to display vital data such as #of fires, how many hours, drought year, etc
        // Before Farsite runs so we can see this on the screen


        printf("\n\t\t*************************************************************\n");
        printf(" \t\t ****                                                \t****\n");
        if (Drought == 1)
        printf(" \t\t ****  Period %d -- This is a WET period.            \t****\n",p);
        else if(Drought == 2)
        printf(" \t\t ****  Period %d --  This is a MODERATE period.      \t****\n",p);
        else if(Drought == 3)
        printf(" \t\t ****  Period %d --  This is a MILD DROUGHT period.  \t****\n",p);
        else
        printf(" \t\t ****  Period %d --  This is a SEVERE DROUGHT period.\t****\n",p);
        printf(" \t\t ****  There will be %d fires this period             \t****\n",NoFires);
        printf(" \t\t ****  Burning for a total of %d hours.              \t****\n",Hours);
        printf(" \t\t ****                                                \t****\n");
        printf(" \t\t **** FARSITE is about to start and may take awhile  \t****\n");
        printf(" \t\t **** Sit back and enjoy the ride.....               \t****\n");
        printf(" \t\t ****                                                \t****\n");
        printf("\t\t*************************************************************\n\n");
}
```

SORT.H

```
/*****************************************************************************
*                                                                            *
*  ------------------------------ sort.h ------------------------------      *
*                                                                            *
*****************************************************************************/

#ifndef SORT_H
#define SORT_H

/*****************************************************************************
*                                                                            *
*  --------------------------- Public Interface ---------------------------  *
*                                                                            *
*****************************************************************************/

int issort(void *data, int size, int esize, int (*compare)(const void *key1,
   const void *key2));

int qksort(void *data, int size, int esize, int i, int k, int (*compare)
   (const void *key1, const void *key2));

int mgsort(void *data, int size, int esize, int i, int k, int (*compare)
   (const void *key1, const void *key2));

int ctsort(int *data, int size, int k);

int rxsort(int *data, int size, int p, int k);

#endif
```

STANDDATA.CPP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "globals.h"                    //to hold global DEFINES, etc..
#include "data.h"

//---------------------------------------------------------------------------------------------------

void StandDataController(struct NEW_STAND_DATA SD[], int Count,  struct TREELIST_RECORD Records[], int NoRecords);
void CalculateIndividualBasalCanopyWidth(struct TREELIST_RECORD Records[], int NoRecords);
void CalculateStandHeight(struct TREELIST_RECORD Records[],int NoRecords,struct NEW_STAND_DATA StandData[], int
Count);
void CalculateStandHLC(struct TREELIST_RECORD Records[], int NoRecords, struct NEW_STAND_DATA StandData[], int
Count);
```

```
void CalculateCBD(struct TREELIST_RECORD Records[], int NoRecords, struct NEW_STAND_DATA StandData[], int Count);



void CalculateStandClassification(struct TREELIST_RECORD Records[], int NoRecords, struct STAND_CLASS *Stand);

int SortTallestTreelistFirst(const void *ptr1, const void *ptr2);
int SortSmallestHlcFirst(const void *ptr1, const void *ptr2);
void RedoHlcCbd(void);

void NewStandHLC( struct STAND_CLASS *Stand  );

//*****************************************************************************************************************
****
void StandDataController(struct NEW_STAND_DATA SD[], int Count,  struct TREELIST_RECORD Records[], int NoRecords)
//*****************************************************************************************************************
****
{
/*
This function will farm out to other functions to calculate various NEW_STAND_DATA for the incoming
TREELIST_RECORD.
"Count" is which line to use in SD[] while Records will hold all the actual treelist records.  Remember, that this
function is being called after an episodic disturbance which create a bunch of snags.  Those snags are not
included in Records[] but that is OK, because these stand metrics being calculated are for live trees only.
*/

int a=0;
struct STAND_CLASS StandClass;
struct STAND_CLASS *ptr_stand;
//------------------------------------ End of variable defining ---------------------------------

//Initialize StandClass and its pointer
ptr_stand = &StandClass;
memset(ptr_stand, 0, sizeof(struct STAND_CLASS) );

//============ REMEMBER:  All these functions are for live trees only ======================================

//First, get the BASAL AREA and CANOPY WIDTH for each record
CalculateIndividualBasalCanopyWidth(Records, NoRecords);

//Get the three items we use in our Veg-Structural classification
CalculateStandClassification(Records, NoRecords, ptr_stand );

//Fill in SD with data returned from ptr_stand
SD[Count].Basal                         = ptr_stand->Basal;
SD[Count].VegClass              = ptr_stand->VegClass;
SD[Count].Qmd                           = ptr_stand->Qmd;
SD[Count].CoverClass            = ptr_stand->CoverClass;
SD[Count].Closure               = ptr_stand->Closure;


//Get the the average Stand Height
CalculateStandHeight(Records, NoRecords, SD, Count);        //NOTE: Records will be descending sorted by "Status"
and "Height" after this

//Get the HLC (or Base to Live Crown)   -- being superceded by NewStandHLC for now 17Feb 00
//CalculateStandHLC(Records, NoRecords, SD, Count );                  //NOTE: Records will be ascending sorted by
"Status" and "Hlc"

//   NEW HLC stuff!!!!!
NewStandHLC(ptr_stand);
SD[Count].HeightCrown           = (ushort)(floor(ptr_stand->HeightCrown + .5));


//Get the Crown Bulk Density for the stand        --
CalculateCBD(Records, NoRecords, SD, Count);


}//end StandDataController


//*****************************************************************************************************************
*
void NewStandHLC( struct STAND_CLASS *Stand  )



//*****************************************************************************************************************
*
{

/*
Use the old matrix that Jim and Bernie developed to classify the HLC based on the stands VegClass and Structural
stage componet - which were already calculated and are in the *Stand structure function.
*/
ushort VegClass, Qmd, CoverClass;
//------------------------------- End of variable defining ---------------------------------------

//Grab values associated with current stand.  It was calculated earlier in CalculateStandClassification()
VegClass  = Stand->VegClass;
Qmd                             = Stand->Qmd;
CoverClass          = Stand->CoverClass;

//printf(" Got %hu %hu %hu here in CalculateNewHLC\n",VegClass,Qmd,CoverClass);
```

```
if( VegClass == VC_DH || VegClass == VC_OPEN)
{
          if( Qmd == 0 )
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 1 )
                    Stand->HeightCrown= (float)(3*M2FT);
          else if(Qmd == 2)
                    Stand->HeightCrown = (float)(3*M2FT);
          else if(Qmd == 3)
          {
                    if(CoverClass == 0 )
                              Stand->HeightCrown = (float)(4*M2FT);
                    else
                              Stand->HeightCrown = (float)(8*M2FT);
          }
          else if(Qmd == 4)
                    Stand->HeightCrown = (float)(8*M2FT);
          else
                    Stand->HeightCrown = (float)(8*M2FT);
}
else if( VegClass == VC_PINE || VegClass == VC_KP )
{
          if( Qmd == 0 )
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 1 )
                    Stand->HeightCrown =(float)(3*M2FT);
          else if(Qmd == 2)
                    Stand->HeightCrown = (float)(6*M2FT);
          else if(Qmd == 3)
                    Stand->HeightCrown = (float)(10*M2FT);
          else if(Qmd == 4)
                    Stand->HeightCrown = (float)(10*M2FT);
          else
                    Stand->HeightCrown = (float)(10*M2FT);
}
else if( VegClass == VC_CH || VegClass == VC_EH )
{
          if( Qmd == 0 )
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 1 )
                    Stand->HeightCrown= (float)(1*M2FT);
          else if(Qmd == 2)
                    Stand->HeightCrown = (float)(1*M2FT);
          else if(Qmd == 3)
                    Stand->HeightCrown = (float)(1*M2FT);
          else if(Qmd == 4)
                    Stand->HeightCrown = (float)(1*M2FT);
          else
                    Stand->HeightCrown = (float)(1*M2FT);
}
else if( VegClass == VC_MC || VegClass == VC_MC3)//I'm cheating here - Jim's matrix shows slight difference but I
"average" for the two
{
          if( Qmd == 0 )
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 1)
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 2)
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 3)
          {
                    if(CoverClass == 0)
                              Stand->HeightCrown  = (float)(1*M2FT);
                    else
                              Stand->HeightCrown  = (float)(7*M2FT);
          }
          else if(Qmd == 4)
                    Stand->HeightCrown  = (float)(1.5*M2FT);
          else
                    Stand->HeightCrown  = (float)(2*M2FT);
}
else if( VegClass == VC_WF )
{
          if(Qmd == 0 )
                    Stand->HeightCrown  = (float)(1*M2FT);
          else if(Qmd == 1)
          {
                    if(CoverClass == 0)
                              Stand->HeightCrown  = (float)(1*M2FT);
                    else
                              Stand->HeightCrown  = (float)(3*M2FT);
          }
          else if(Qmd == 2)
          {
                    if(CoverClass == 0)
                              Stand->HeightCrown  = (float)(1*M2FT);
                    else
                              Stand->HeightCrown  = (float)(4*M2FT);
          }
          else if( Qmd == 3)
          {
                    if(CoverClass == 0)
                              Stand->HeightCrown  = (float)(1*M2FT);
                    else
```

```
                        Stand->HeightCrown  = (float)(7*M2FT);
        }
        else if( Qmd == 4)
        {
                if(CoverClass == 0)
                        Stand->HeightCrown  = (float)(2*M2FT);
                else
                        Stand->HeightCrown  = (float)(3*M2FT);
        }
        else
                Stand->HeightCrown = (float)(10*M2FT);
}
else                                                                    //should be for VC_RF only
{
        if(Qmd == 0 )
                Stand->HeightCrown= (float)(1*M2FT);
        else if(Qmd == 1)
        {
                if(CoverClass == 0)
                        Stand->HeightCrown  = (float)(1*M2FT);
                else
                        Stand->HeightCrown  = (float)(5*M2FT);
        }
        else if(Qmd == 2)
        {
                if(CoverClass == 0)
                        Stand->HeightCrown  = (float)(1*M2FT);
                else
                        Stand->HeightCrown  = (float)(3*M2FT);
        }
        else if( Qmd == 3)
        {
                if(CoverClass == 0)
                        Stand->HeightCrown  = (float)(1*M2FT);
                else
                        Stand->HeightCrown= (float)(7*M2FT);
        }
        else if( Qmd == 4)
        {
                if(CoverClass == 0)
                        Stand->HeightCrown  = (float)(10*M2FT);
                else
                        Stand->HeightCrown  = (float)(1*M2FT);
        }
        else
                Stand->HeightCrown = (float)(10*M2FT);
}

}//end NewStandHLC


//*****************************************************************************************************************
*
void CalculateCBD(struct TREELIST_RECORD Records[], int NoRecords, struct NEW_STAND_DATA StandData[], int Count)



//*****************************************************************************************************************
*
{
/*
CBD calculations were in PREMO from stuff John put in based on something Jim A. gave us.
Will try and copy what he did here.
*/

int a;
double StandCBD=0;
float Dbh, Tpa;
double CbdCF, CbdM3;
double ModCbd;
//-------------------- End of variable defining ----------------------------------------------------


//Calculate the entire stand CBD as a function of the individual species and equations for each
for(a=0;a<NoRecords;a++)
{
        if(Records[a].Status == LIVE )
        {
                //Set the Dbh & Tpa for ease of reading
                Dbh = Records[a].Dbh;
                Tpa = Records[a].Tpa;


                if(Records[a].Model             == BLACKOAK)
                                                                                                        //0
                        StandCBD += 0.8 * Dbh * Tpa;
                else if(Records[a].Model == DOUGFIR)
                                                                                                //1
                        StandCBD += exp(-2.8462+1.7009*log(2.54*Dbh))*Tpa*2.2046;
                else if(Records[a].Model == ICEDAR)
                                                                                                        //2
                        StandCBD += exp(-2.617+1.7824*log(2.54*Dbh))*Tpa*2.2046;
```

```
                else if(Records[a].Model == KPINE || Records[a].Model == PPINE)
                                                                        //3 or 5
                        StandCBD += exp(-4.2612+2.0967*log(2.54*Dbh))*Tpa*2.2046;
                else if(Records[a].Model == MADRONE)
                                                                                        //4
                        StandCBD += 1.4*Dbh*Tpa;
                else if(Records[a].Model == RFIR || Records[a].Model == WFIR )
                                                                        //6 or 9
                        StandCBD += exp(-3.4662+1.9278*log(2.54*Dbh))*Tpa*2.2046;
                else if(Records[a].Model == SPINE)
                                                                                        //7
                        StandCBD += exp(-3.9739+2.0039*log(2.54*Dbh))*Tpa*2.2046;
                else if(Records[a].Model == TANOAK)
                                                                                        //8
                        StandCBD += 5.0*Dbh*Tpa;
                else

                                //all others
                        StandCBD += (400./40.)*Dbh*Tpa;
        }

}//end for(a=0 ...)


//Calculate the CBD per CF first;
CbdCF     = StandCBD / ( (StandData[Count].StandHeight - StandData[Count].HeightCrown) * 43560 );
                // in lbs/ft3

if(CbdCF < 0 )
  CbdCF = 0;

//Then calculate the CBD in kg per m3
CbdM3     = CbdCF * 16.02;
                                        // kg per m3

//Store the kg per m3 data in Stand
//StandData[Count].Density    = (ushort)(floor(CbdM3*DENSITY_EXP) );

//*****************************************************************************
//  Below is the Bernie "tweak"                         17Feb00
//*****************************************************************************
ModCbd = CbdM3 * ( StandData[Count].Closure * ACREEQ) ;

if( ModCbd > .30 )
        ModCbd = .30;

//Store the kg per m3 data in Stand
StandData[Count].Density      = ((ushort)(ModCbd*DENSITY_EXP));


}//end CalculateCBD



//*****************************************************************************************************************
****
void CalculateStandHLC(struct TREELIST_RECORD Records[], int NoRecords, struct NEW_STAND_DATA StandData[], int
Count)
//*****************************************************************************************************************
****
{
int a;
double StandTpa=0, StandHlc;
double Threshold50, MidStoryThreshold, MidStoryTpa;
double Group;
double SumTpa=0;
int Tpa50, Tpa50Tree, Flag;
//---------------------- End of variable defining ----------------------------------------

//Create an array to hold the HtlcGroup for the records - and initialize
int (*HlcGroup) = new int[NoRecords];
if( HlcGroup == NULL)
        printf("Problems allocating memory for HlcGroup with %lu elements\n",NoRecords);
memset(HlcGroup, 0, sizeof(*HlcGroup) * NoRecords);


//Fill the Records[].Hlc member and calculate the StandTpa
for(a=0;a<NoRecords;a++)
{
        if( Records[a].Status == LIVE)
        {
                Records[a].Hlc      =  Records[a].Height - ( (Records[a].Height * Records[a].Ratio) / 100 );
                StandTpa            += Records[a].Tpa;
        }
}

//The Hlc member needs to be sorted with smallest values first
qsort(  (void*)Records,
                        //base
       (size_t)NoRecords,
                //count of records
                sizeof(struct TREELIST_RECORD) ,
         //size of each record
                SortSmallestHlcFirst );
                        //compare function
```

```
//test print
/*
if(Count == 0)
{
printf("The Hlc member AFTER sorting\n");
for(a=0;a<NoRecords;a++)
          printf("Records[%d].Hlc is %.2lf and .Tpa is %.2lf\n",a,Records[a].Hlc, Records[a].Tpa);
}
*/

//Set some threshold values
Threshold50         = StandTpa * 0.1;                                   //I don't know why
if(Threshold50 > 50 )
          Threshold50 = 50;

//printf("\nThreshold50 just set to %.3lf\n\n",Threshold50);

MidStoryThreshold   = StandTpa * 0.05;            //I don't know why
if(MidStoryThreshold > 5 )
          MidStoryThreshold = 5;


//Fill the HtlcGroup array
for(a=0;a<NoRecords;a++)
{
          Group = Records[a].Hlc * 7.62 / 25 ;              //no ideal where equation came from
          HlcGroup[a] = (int)(floor(Group + 0.5));
}


//Now sum of Tpa for while threshold is not exceeded                  - no ideal why
a=0;
while(SumTpa < Threshold50 )            //Start looking from the top
{
          SumTpa += Records[a].Tpa;
          if( SumTpa < Threshold50 )
                    a++;
}

//Get the next HtlcGroup after above threshold violated - and set a marker for that tree
Tpa50 = HlcGroup[a];
Tpa50Tree = a;
//printf("Tpa50 set to %d with Tpa50Tree at %d\n",Tpa50, Tpa50Tree);

Flag=0;
while( Flag == 0 )
{
          //Doing something here
          MidStoryTpa = 0;
          for(a=0;a<NoRecords;a++)
          {
                    if( HlcGroup[a] > Tpa50 && HlcGroup[a] < Tpa50+4 )                    //Look for a 3' increment
above Tpa50
                              MidStoryTpa += Records[a].Tpa;
          }


          //Doing something here
          if( MidStoryTpa > MidStoryThreshold)
          {
                    Flag = 1;
                    StandHlc = Records[Tpa50Tree].Hlc;
          }
          else
          {
                    a=0;
                    Tpa50++;                        //increment up 1'

                    while( Records[a].Hlc < (Tpa50 * 25 / 7.62)   &&     a < (NoRecords-1) )
                              a++;

                    Tpa50Tree = a;
          }

          //Doing something here
          if( Tpa50 > HlcGroup[NoRecords-1] )
          {
                    Flag =1;
                    StandHlc = Records[Tpa50Tree].Hlc;
          }
}//end while

//printf("Just got a stand HLC of %.3lf\n",StandHlc);


//Store in the Stand structure
StandData[Count].HeightCrown = (ushort)( floor(StandHlc+0.5) );


//Delete stuff on free store
delete [] HlcGroup;

}//end CalculateStandHLC
```

```
//*********************************************************************
int SortSmallestHlcFirst(const void *ptr1, const void *ptr2)
//*********************************************************************
{

          //Just to typecast them since we aren't actually passing in pointers
          struct TREELIST_RECORD *elem1;
          struct TREELIST_RECORD *elem2;

          elem1 = (struct TREELIST_RECORD *)ptr1;
          elem2 = (struct TREELIST_RECORD *)ptr2;

    if( elem1->Status < elem2->Status )                                                    //Sort by
Status first
        return -1;
    if( elem1->Status > elem2->Status )
        return 1;
    else
          {
                    if( elem1->Hlc < elem2->Hlc )                                          //Then sort by
Height to Live Crown
                              return -1;
                    if( elem1->Hlc > elem2->Hlc )
                              return 1;
                    else
                              return 0;
                    //Finished
          }

}//end SortSmallestHlcFirst


//*********************************************************************************************************************
****
void CalculateStandHeight(struct TREELIST_RECORD Records[],int NoRecords,struct NEW_STAND_DATA StandData[], int
Count)
//*********************************************************************************************************************
****
{
/*
Calculate the average stand height by:   ??????

Has something to do with finding the records for those largest records whose TPA average out to something
like NO_TALL_TREES (using 5 at first).  I think Jim A. came up with this strategy
*/

int a;
float TallTpa=0, HeightTallTrees=0, Tallest=0;
//------------------------------------ End of variable defining ------------------------------------

//First, sort Records by height, with the Tallest trees first (and broken into LIVE, DWD, and SNAG)
qsort(  (void*)Records,
                              //base
        (size_t)NoRecords,
                    //count of records
                    sizeof( struct TREELIST_RECORD ),
                    //size of each record
                    SortTallestTreelistFirst );
                              //compare function


a=0;
while( TallTpa < NO_TALL_TREES && a < NoRecords )
{
          if(Records[a].Status == LIVE)
          {

                    TallTpa                       += Records[a].Tpa;
                    HeightTallTrees += Records[a].Height * Records[a].Tpa;

                    if( TallTpa > NO_TALL_TREES )
                              HeightTallTrees     -= Records[a].Height * (TallTpa-NO_TALL_TREES);
          }

          a++;
}//end while( ...


//Now average those trees out
if(TallTpa > 0 )
          HeightTallTrees /= NO_TALL_TREES;
else
          HeightTallTrees = 0;

//Store in the Stand structure
StandData[Count].StandHeight = (ushort)( floor(HeightTallTrees+0.5) );


}//end CalculateStandHeight

//*********************************************************************
int SortTallestTreelistFirst(const void *ptr1, const void *ptr2)
//*********************************************************************
```

```
{

          //Just to typecast them since we aren't actually passing in pointers
          struct TREELIST_RECORD *elem1;
          struct TREELIST_RECORD *elem2;

          elem1 = (struct TREELIST_RECORD *)ptr1;
          elem2 = (struct TREELIST_RECORD *)ptr2;


    if( elem1->Status > elem2->Status )                                                    //Sort by
Status first
        return -1;
    if( elem1->Status < elem2->Status )
        return 1;
    else
          {
                    if( elem1->Height > elem2->Height )
          //Then by height
                              return -1;
                    if( elem1->Height < elem2->Height )
                              return 1;
                    else
                              return 0;
                                        //FINISHED
          }

}//end SortTallestTreelistFirst


//*****************************************************************************************************************
****
void CalculateIndividualBasalCanopyWidth(struct TREELIST_RECORD Records[], int NoRecords)
//*****************************************************************************************************************
****
{
/*
This function will do two things:

1 - Calculate a Basal Area for each live tree record
2 - Calculate the canopy width for each live tree record

Not sure where PREMO got these canopy width coefficients but I am copying them straight from PREMO.

It appears that canopy width is a function of height ( > or < 4.5 feet), some
coefficient based on the Model species code, and either Dbh or Height again.
*/
int a;
ushort SpeciesCode;
float Height, Dbh;

//Now idea where these came from!
double CO1[TOTALSP]={2.4922,4.4215,4.0920,2.8541,7.5183,2.8541,3.1146,3.2367,7.5183,3.8166};
double CO2[TOTALSP]={0.8544,0.5329,0.4912,0.6400,0.4461,0.6400,0.5780,0.6247,0.4461,0.5229};
double CO3[TOTALSP]={0.1400,0.5170,0.4120,0.4070,0.8150,0.4070,0.3450,0.4060,0.8150,0.4520};

//------------------------------End of variable defining -------------------

//Look at each record and calculate BA by formula in Forestry Handbook
for(a=0;a<NoRecords;a++)
{
          if( Records[a].Status == LIVE )
          {
                    //Set some variable for ease
                    Dbh                                   = Records[a].Dbh;
                    Height                          = Records[a].Height;
                    SpeciesCode                     = Records[a].Model;                    //Use the modeling code
(value 0-9 to use in array subscript)


                    //Calculate Basal area and put in the current record
                    Records[a].Basal    = (float)( pow(Dbh,2) * BASAL_CONSTANT);

                    //Calculate Canopy Width and put in the current record
                    if(Height > 4.5)
                              Records[a].CanopyWidth        = (float)( CO1[SpeciesCode] * pow(Dbh,CO2[SpeciesCode])
);
                    else
                              Records[a].CanopyWidth        = (float)( CO3[SpeciesCode] * Height );

          }//end if(Status == LIVE)

}//end for(a=0 ...)


}//end CalculateBasalCanopyWidth

//*****************************************************************************************************************
**************
void CalculateStandClassification(struct TREELIST_RECORD Records[], int NoRecords, struct STAND_CLASS *Stand)
//*****************************************************************************************************************
**************
{
/*
```

```
The *Stand (pointer) gets filled with data and is sent back without regards to what structure it will be going
into.

The equations were taken from PREMO and I believe they come from the work Lou Beers outlined in his paper
"Methods used to calculate QMD, VegType, and percent canopy closure on the Applegate watershed" 8-24-98

NOTE:
#define BLACKOAK              0                          //used as array subscripts
#define DOUGFIR                        1
#define ICEDAR                         2
#define KPINE                          3
#define MADRONE                        4
#define PPINE                          5
#define RFIR                           6
#define SPINE                          7
#define   TANOAK                       8
#define WFIR                           9

NOTE:  sometimes this functions gets called and only LiveRecords are passed in - sometimes not.  So to be safe,
always check
that calculations are done for LIVE trees only.
*/

int a;
double TempCover, TotalCover=0, StandBa=0, AdjStandBa=0, StandTpa=0, AdjStandTpa=0, Qmd, AdjQmd;
double SpCover[TOTALSP];
double RealStandBasal=0;
//---------------------------- End of variable definition ---------------------------------

//Initialize SpCover (SpeciesCover)
for(a=0;a<TOTALSP;a++)
          SpCover[a] = 0;

//============================================ Get the average Qmd and stand BASAL ================================
//Get an average stand Qmd
for(a=0;a<NoRecords;a++)
{
          //Calculate the RealStandBasal for live trees only
          if( Records[a].Status == LIVE )
          {
                    RealStandBasal += Records[a].Basal * Records[a].Tpa;

                    if (Records[a].Dbh >=1 )                         //Only those larger than 1" contribute when
calculate Vegclass only!
                    {
                              StandBa             += Records[a].Basal * Records[a].Tpa;
                              StandTpa  += Records[a].Tpa;
                    }
          }
}

Qmd = pow( (StandBa / (BASAL_CONSTANT*StandTpa)), 0.5);

//Put the RealStandBasal in the Stand structure
Stand->Basal = (float)RealStandBasal;


//======================================== Get the AdjQmd  ================================

//Go through records again and only count those that are larger than the above determined Qmd
for(a=0;a<NoRecords;a++)
{
          if( Records[a].Status == LIVE)
          {
                    if (Records[a].Dbh >= Qmd )
                    {
                              AdjStandBa                                                  += Records[a].Basal *
Records[a].Tpa;
                              AdjStandTpa                                                 += Records[a].Tpa;
                              TempCover                                         =  pow((Records[a].CanopyWidth/2)
,2) * PI * Records[a].Tpa;          //only calculate once
                              TotalCover                                                  += TempCover;
                              SpCover[Records[a].Model]     += TempCover;
                    }
          }
}

AdjQmd = pow( (AdjStandBa / (BASAL_CONSTANT*AdjStandTpa)), 0.5);

//======================================== Get the Canopy Closure ================================
//Not sure what PREMO is doing to the SpCover here - some compensation for sq ft. per acre or something
for(a=0;a<TOTALSP;a++)
          SpCover[a]          = (SpCover[a] / TotalCover)*100;

//Adjust TotalCover for acres -
TotalCover = TotalCover / 435.6;                                                      //43,560 / 100%

//printf("Just got a TotalCover of %.3lf\n",TotalCover);

//Put the TotalCover in Stand->Closure (this is the canopy closure percentage)
Stand->Closure = (ushort)TotalCover;


//================================== Get the vegclassification code ================================
//Get the category which is used as our vegetation category for mapping and GIS stuff
```

```
//The following will try and copy what PREMO had, but may be slightly different to ease reading and coding
if(       SpCover[BLACKOAK] +
    SpCover[DOUGFIR]  +
    SpCover[ICEDAR]   +
    SpCover[KPINE]    +
    SpCover[MADRONE]  +
    SpCover[PPINE]    +
    SpCover[RFIR]     +
    SpCover[SPINE]    +
    SpCover[TANOAK]   +
    SpCover[WFIR]                                          < 20 )
                Stand->VegClass = VC_OPEN;
                                                    //Open (?)
else if(SpCover[BLACKOAK]  + SpCover[MADRONE]   +  SpCover[TANOAK]    > 30 )
{
          if(
                SpCover[DOUGFIR]  +
                SpCover[ICEDAR]   +
                SpCover[KPINE]    +
                SpCover[PPINE]    +
                SpCover[RFIR]     +
                SpCover[SPINE]    +
                SpCover[WFIR]                                      > 30 )
                          Stand->VegClass = VC_CH;
                                                    //CH
          else if( SpCover[BLACKOAK]  + SpCover[TANOAK]     > 50 )
                          Stand->VegClass = VC_EH;
                                                    //EH
          else
                          Stand->VegClass = VC_DH;
                                                    //DH
}
else if( SpCover[RFIR] + SpCover[WFIR]  > 50 )
{
          if( SpCover[RFIR]  > SpCover[WFIR] )
                Stand->VegClass = VC_RF;
                                                    //RF
          else
                Stand->VegClass = VC_WF;
                                                    //WF
}
else if( SpCover[KPINE] + SpCover[PPINE] + SpCover[SPINE]   < 50 )
          Stand->VegClass = VC_MC;
                                                    //MC
else if( SpCover[PPINE] + SpCover[SPINE] > SpCover[KPINE] )
          Stand->VegClass = VC_PINE;
                                          //Pine
else
          Stand->VegClass     = VC_KP;
                                          //KP


//============================== Get the CoverClass category used as part of our structural stage
==========================
if(TotalCover < 60 )
          Stand->CoverClass = 0;
else
          Stand->CoverClass = 1;


//=================================== Get the QMD category that is also used as part of our structural stage
==============
if(AdjQmd < 5)
          Stand->Qmd          = 0;
else if(AdjQmd < 9)
          Stand->Qmd          = 1;
else if(AdjQmd < 15)
          Stand->Qmd          = 2;
else if(AdjQmd < 21)
          Stand->Qmd          = 3;
else if(AdjQmd < 25)
          Stand->Qmd          = 4;
else if(AdjQmd < 32)
          Stand->Qmd          = 5;
else
          Stand->Qmd          = 6;

//Readjust the CoverClass for those stands that don't have a cover component because of the QMD (the real young and
old stands)
if(Stand->Qmd == 0 ||Stand->Qmd == 6 || Stand->Qmd == 5 )
          Stand->CoverClass = 0;

}//end CalculateStandClassification


//*****************************************************************************************************************
****
void RedoHlcCbd(void)
//*****************************************************************************************************************
****
{
/*
17Feb00 - Decided by Bernie and I that the HLC and CBD values coming from Premo were just not working.  We
decided to try and use the old way of calculating HLC.  The old way (for HLC) is by using the Matrix that Jim Agee
and Bernie originally developed to classify based on the stands VegClass and Structural Stage.  The
```

```
CBD measurements are more complicated and for now we are going to just "tweak" the values that are
being generated in Premo.

Values in Data.Vegcode are those 3 or 4 digits values that were either generated directly in PREMO or were slightly
modified by this progrom in FillPremoData().  Heidi gave me the following regarding what the PREMO codes meant:

1st digit = (veg. class)
1         CH                                                    #define VC_CH
2         DH                                                    #define VC_DH
3         EH                                                    #define VC_EH
4         CCP                                                   #define VC_KP
5         MC                                                    #define VC_MC
6         open ????                                 #define VC_OPEN
7         Pine                                              #define VC_PINE
8         RF                                                    #define VC_RF
9         WF                                                    #define VC_WF
(10)Not used in Premo                    #define VC_MC3


2nd digit = (QMD)
0         0-4.9
1         5-8.9
2         9-14.9
3         15-20.9
4         21-24.9
5         25-31.9
6         32+

3rd digit = (Canopy closure)
0         <= 60%
1         > 60%

Alterations: FillInitialPremoData() changed those with an original 1st digit of 5 to be either 5 (MC < 3000') or 10
(MC > 10000'),
so I can directly check for 5 or 10.
*/

int a, b;
ushort TempCode;
int TempVeg, TempDiam, TempCover;
float ModCbd;
//------------------------------ End of variable defining ------------------------------------

printf("Recalculating the HLC and CBD with different algorithms than used in PREMO\n");


//Go through all of Data.*[]
for(a=0;a<UNIQUE;a++)
{
          if(Data.Cellid[a] == FALSE )  //no more cells to check
                    break;

          if(Data.Treelist[a] == NONFOREST )
                    continue;

          for(b=0;b<NP;b++)
          {

                    //*****************************************************************************************
                    //                                                                    Do the new HLC
                    //*****************************************************************************************
                    TempCode = Data.Vegcode[a][b];                                    //The actual 3 or 4 digit code from
PREMO

                    //extract the digits out
                    TempCover = TempCode%10;                                                                  //last digit
for determining stage (is closure, <=60% or > 60% )
                    TempDiam = ( (TempCode-TempCover)%100 ) / 10;     //next to last digit also for determining
stage (is the QMD group)
                    TempVeg = (TempCode-TempCode%100) / 100;                    //1st or 1st two digits for
determining VegCode



                    if( TempVeg == VC_DH || TempVeg == VC_OPEN)
                    {
                              if( TempDiam == 0 )
                                        Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                              else if(TempDiam == 1 )
                                        Data.HLC[a][b] = (ushort)(floor(3*M2FT + .5));
                              else if(TempDiam == 2)
                                        Data.HLC[a][b] = (ushort)(floor(3*M2FT + .5));
                              else if(TempDiam == 3)
                              {
                                        if(TempCover == 0 )
                                                  Data.HLC[a][b] = (ushort)(floor(4*M2FT + .5));
                                        else
                                                  Data.HLC[a][b] = (ushort)(floor(8*M2FT + .5));
                              }
                              else if(TempDiam == 4)
                                        Data.HLC[a][b] = (ushort)(floor(8*M2FT + .5));
                              else
                                        Data.HLC[a][b] = (ushort)(floor(8*M2FT + .5));
                    }
                    else if( TempVeg == VC_PINE || TempVeg == VC_KP )
```

```
{
        if( TempDiam == 0 )
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 1 )
                Data.HLC[a][b]= (ushort)(floor(3*M2FT + .5));
        else if(TempDiam == 2)
                Data.HLC[a][b] = (ushort)(floor(6*M2FT + .5));
        else if(TempDiam == 3)
                Data.HLC[a][b] = (ushort)(floor(10*M2FT + .5));
        else if(TempDiam == 4)
                Data.HLC[a][b] = (ushort)(floor(10*M2FT + .5));
        else
                Data.HLC[a][b] = (ushort)(floor(10*M2FT + .5));
}
else if( TempVeg == VC_CH || TempVeg == VC_EH )
{
        if( TempDiam == 0 )
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 1 )
                Data.HLC[a][b] = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 2)
                Data.HLC[a][b] = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 3)
                Data.HLC[a][b] = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 4)
                Data.HLC[a][b] = (ushort)(floor(1*M2FT + .5));
        else
                Data.HLC[a][b] = (ushort)(floor(1*M2FT + .5));
}
else if( TempVeg == VC_MC  || TempVeg == VC_MC3 )
{
        if( TempDiam == 0 )
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 1)
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 2)
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 3)
        {
                if(TempCover == 0)
                        Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                else
                        Data.HLC[a][b]      = (ushort)(floor(7*M2FT + .5));
        }
        else if(TempDiam == 4)
                Data.HLC[a][b]      = (ushort)(floor(1.5*M2FT + .5));
        else
                Data.HLC[a][b]      = (ushort)(floor(2*M2FT + .5));
}
else if( TempVeg == VC_WF )
{
        if(TempDiam == 0 )
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 1)
        {
                if(TempCover == 0)
                        Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                else
                        Data.HLC[a][b]      = (ushort)(floor(3*M2FT + .5));
        }
        else if(TempDiam == 2)
        {
                if(TempCover == 0)
                        Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                else
                        Data.HLC[a][b]      = (ushort)(floor(4*M2FT + .5));
        }
        else if( TempDiam == 3)
        {
                if(TempCover == 0)
                        Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                else
                        Data.HLC[a][b]      = (ushort)(floor(7*M2FT + .5));
        }
        else if( TempDiam == 4)
        {
                if(TempCover == 0)
                        Data.HLC[a][b]      = (ushort)(floor(2*M2FT + .5));
                else
                        Data.HLC[a][b]      = (ushort)(floor(3*M2FT + .5));
        }
        else
                Data.HLC[a][b] = (ushort)(floor(10*M2FT + .5));
}
else
        //should be for VC_RF only
{
        if(TempDiam == 0 )
                Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
        else if(TempDiam == 1)
        {
                if(TempCover == 0)
                        Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                else
                        Data.HLC[a][b]      = (ushort)(floor(5*M2FT + .5));
```

```
                    }
                    else if(TempDiam == 2)
                    {
                            if(TempCover == 0)
                                    Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                            else
                                    Data.HLC[a][b]      = (ushort)(floor(3*M2FT + .5));
                    }
                    else if( TempDiam == 3)
                    {
                            if(TempCover == 0)
                                    Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                            else
                                    Data.HLC[a][b]      = (ushort)(floor(7*M2FT + .5));
                    }
                    else if( TempDiam == 4)
                    {
                            if(TempCover == 0)
                                    Data.HLC[a][b]      = (ushort)(floor(10*M2FT + .5));
                            else
                                    Data.HLC[a][b]      = (ushort)(floor(1*M2FT + .5));
                    }
                    else
                            Data.HLC[a][b] = (ushort)(floor(10*M2FT + .5));
            }


            //*****************************************************************************************
            //                                                                  Do the new CBD
            //*****************************************************************************************

            ModCbd = (float)( ( (float)Data.CBDensity[a][b] / DENSITY_EXP ) * (Data.Closure[a][b] * ACREEQ)
);

            if( ModCbd > .30 )
                    Data.CBDensity[a][b] = (ushort)(.30 * DENSITY_EXP);
            else
                    Data.CBDensity[a][b] = (ushort)(ModCbd * DENSITY_EXP);
        }
}//end for(a=0... )

}//end RedoHlcCbd
```